

# **TRICKS OF THE 3D GAME PROGRAMMING GURUS**

**ADVANCED 3D GRAPHICS AND  
RASTERIZATION**

*Andre LaMothe*

**SAMS**

201 West 103rd Street,  
Indianapolis, Indiana 46230

# **ПРОГРАММИРОВАНИЕ ТРЕХМЕРНЫХ ИГР ДЛЯ WINDOWS**

**СОВЕТЫ ПРОФЕССИОНАЛА ПО  
ТРЕХМЕРНОЙ ГРАФИКЕ И РАСТЕРИЗАЦИИ**

# ПРОГРАММИРОВАНИЕ ТРЕХМЕРНЫХ ИГР ДЛЯ WINDOWS

СОВЕТЫ ПРОФЕССИОНАЛА ПО ТРЕХМЕРНОЙ  
ГРАФИКЕ И РАСТЕРИЗАЦИИ

*Андре Ламот*



Москва • Санкт-Петербург • Киев

2004

ББК 32.973.26-018.2.75

Л21

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Р.Г. Имамудиновой*, канд. техн. наук *И.В. Красикова*,  
*А. Наумовца, Н.А. Ореховой, В.Н. Романова*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:  
info@williamspublishing.com, <http://www.williamspublishing.com>

**Ламот, Андре.**

Л21 Программирование трехмерных игр для Windows. Советы профессионала по трехмерной графике и растеризации. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 1424 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0627-X (рус.)

Данная книга представляет собой продолжение книги Андре Ламота *Программирование игр для Windows. Советы профессионала* и посвящена созданию трехмерных игр. В книге освещены различные аспекты разработки трехмерных игр, однако основное внимание уделяется вопросам программирования трехмерных игр — в частности, представления трехмерных объектов, их визуализации с учетом свойств материала объектов, освещения, перспективы, а также таким специфическим вопросам трехмерной визуализации, как создание различных визуальных спецэффектов и т.п. В книге также рассматриваются многие сопутствующие вопросы — создание и применение звуковых эффектов и музыкального сопровождения, использование различных форматов файлов и соответствующего инструментария.

Книга написана выдающимся специалистом в области программирования игр с многолетним стажем, и полезна как начинающим, так и профессиональным разработчикам игр для Windows. Однако следует учесть, что она рассчитана в первую очередь на опытного специалиста, владеющего языком программирования C++, а также имеющего определенную математическую подготовку. Хотя данная книга может рассматриваться как отдельное издание, желательно приступить к ней после ознакомления с упомянутой ранее книгой.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2003

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2004

ISBN 5-8459-0627-X (рус.)

ISBN 0-672-31835-0 (англ.)

© Издательский дом "Вильямс", 2004

© by Sams Publishing, 2003

# Оглавление

Предисловие	17
Об авторе	18
О техническом редакторе	18
Благодарности	19
Введение	21
<b>Часть I. Введение в программирование трехмерных игр</b>	<b>27</b>
Глава 1. Основы программирования трехмерных игр	29
Глава 2. Краткий курс Windows и DirectX	79
Глава 3. Виртуальный компьютер для программирования трехмерных игр	119
<b>Часть II. Трехмерная математика и преобразования</b>	<b>219</b>
Глава 4. Запутанный мир математики	221
Глава 5. Создание математической библиотеки	309
Глава 6. Введение в трехмерную графику	409
Глава 7. Визуализация трехмерных каркасных объектов	515
<b>Часть III. Основы трехмерной визуализации</b>	<b>627</b>
Глава 8. Основы моделирования освещения и поверхностей тел	629
Глава 9. Интерполяционные методы затенения и аффинное отображение текстур	749
Глава 10. Отсечение в трехмерном пространстве	879
Глава 11. Организация буфера глубины и видимость	957
<b>Часть IV. Секреты трехмерной визуализации</b>	<b>1013</b>
Глава 12. Методы сложного текстурирования	1015
Глава 13. Алгоритмы разбиения пространства и определения видимости	1143
Глава 14. Освещение и тени	1263
<b>Часть V. Анимация, физическое моделирование и оптимизация</b>	<b>1309</b>
Глава 15. Анимация, движение и обнаружение столкновений	1311
Глава 16. Технологии оптимизации	1359
Предметный указатель	1407

# Содержание

Предисловие	17
Об авторе	18
О техническом редакторе	18
Благодарности	19
Введение	21
<b>Часть I. Введение в программирование трехмерных игр</b>	<b>27</b>
<b>Глава 1. Основы программирования трехмерных игр</b>	<b>29</b>
Краткое введение	30
Элементы двумерных и трехмерных игр	31
Общие советы по программированию игр	36
Использование инструментов	40
Редакторы трехмерных уровней	43
Использование компилятора	44
Пример трехмерной игры: Raiders 3D	48
Цикл событий	73
Внутренняя логика игры	73
Трехмерные проекции	75
Звездное поле	77
Лазерные пушки и обнаружение попаданий	77
Взрывы	77
Как играть в Raiders3D	77
Резюме	78
<b>Глава 2. Краткий курс Windows и DirectX</b>	<b>79</b>
Модель программирования Win32	80
Минимальный курс программирования для Windows	81
Все начинается с WinMain()	81
Базовое приложение для Windows	87
Класс Windows	87
Регистрация класса Windows	92
Создание окна	93
Обработчик событий	95
Главный цикл событий	100
Создание цикла событий реального времени	104
Краткий курс DirectX и COM	106
HEL и HAL	107
Подробнее о базовых классах DirectX	108
Краткое введение в COM	110
Что такое COM-объект?	111
Создание и использование COM-интерфейсов DirectX	114

Запрос интерфейсов	115
Резюме	116
<b>Глава 3. Виртуальный компьютер для программирования трехмерных игр</b>	119
Введение в интерфейс виртуального компьютера	120
Построение интерфейса виртуального компьютера	122
Буфер кадра и видеосистема	122
Работа с цветом	126
Анимация	128
Полная виртуальная графическая система	130
Ввод-вывод, звук и музыка	131
Консоль игры T3DLIB	132
Обзор T3DLIB	132
Базовая консоль игры	132
Библиотека T3DLIB1	138
Архитектура графического процессора DirectX	139
Основные определения	139
Макросы	141
Типы и структуры данных	142
Прототипы функций	145
Глобальные переменные	151
Интерфейс DirectDraw	152
Функции для работы с двумерными многоугольниками	156
Двумерные графические примитивы	159
Математические функции и функции обработки ошибок	163
Функции для работы с растровыми изображениями	165
Функции для работы с 8-битовыми палитрами	169
Вспомогательные функции	172
Процессор для работы с объектами блиттера	174
Система ввода DirectX	182
Звуковая и музыкальная библиотека T3DLIB3	188
Заголовочный файл	189
Типы	189
Глобальные переменные	190
API оболочки DirectSound	190
API оболочки DirectMusic	194
Окончательная версия консоли игры	197
Графика — реальная и виртуальная	197
Консоль игры — окончательный вариант	200
Образцы приложений T3DLIB	211
Оконные приложения	211
Полноэкранное приложение	211
Звук и музыка	213
Работа с устройствами ввода	213
Резюме	216
 <b>Часть II. Трехмерная математика и преобразования</b>	 219
<b>Глава 4. Запутанный мир математики</b>	221
Математические обозначения	222

Двумерные системы координат	223
Двумерные декартовы координаты	223
Двумерные полярные координаты	225
Трёхмерные системы координат	228
Трёхмерные декартовы координаты	229
Трёхмерные <b>цилиндрические</b> координаты	230
Трёхмерные сферические <b>координаты</b>	233
Тригонометрия	234
Прямоугольный треугольник	235
Обратные тригонометрические функции	237
Тригонометрические тождества	237
Векторы	238
Длина вектора	239
Нормализация	240
Умножение на скаляр	240
Сложение векторов	241
Вычитание векторов	241
Скалярное произведение	242
Векторное умножение	245
Нулевой вектор	246
Радиус-вектор и вектор <b>перемещения</b>	246
Векторы как линейные комбинации	247
Матрицы и <b>линейная</b> алгебра	247
Единичная матрица	249
Сложение матриц	250
Транспонирование матрицы	250
Умножение матриц	250
Свойства матричных операций	252
Обращение матриц и решение систем линейных уравнений	252
Правило Крамера	253
Преобразования с использованием матриц	255
Однородные координаты	256
Применение матричных преобразований	257
Фундаментальные геометрические объекты	263
Точки	263
Прямые линии	263
Прямые в трёхмерном пространстве	265
Плоскости	267
Использование параметрических <b>уравнений</b>	271
Двумерные и трёхмерные параметрические прямые	271
Параметрическое представление отрезка с помощью стандартного вектора направления	272
Параметрическое представление отрезка с помощью единичного вектора направления	273
Параметрическое представление трёхмерных прямых	273
Вычисление пересечения параметризованных прямых	274
Вычисление пересечения отрезка и плоскости	277
Введение в кватернионы	278
Теория комплексных чисел	278

Гиперкомплексные числа	282
Применение кватернионов	287
Дифференциальное исчисление	291
Концепция бесконечности	291
Пределы	293
Суммы и конечные ряды	294
Бесконечные ряды	296
Производные	297
Интегралы	303
Резюме	308
<b>Глава 5. Создание математической библиотеки</b>	<b>309</b>
Краткий обзор математической библиотеки	310
Структура математической библиотеки	310
Соглашения об именах	311
Обработка ошибок	312
Заключительное слово о C++	312
Типы и структуры данных	313
Векторы и точки	313
Параметризованные прямые	314
Трехмерные плоскости	316
Матрицы	317
Кватернионы	320
Угловые системы координат	321
Двумерные полярные координаты	322
Трехмерные цилиндрические координаты	322
Трехмерные сферические координаты	323
Числа с фиксированной точкой	323
Математические константы	324
Макросы и встраиваемые функции	327
Утилиты общего назначения и преобразование величин	331
Точки и векторы	332
Матрицы	333
Кватернионы	335
Математика с фиксированной точкой	336
Прототипы	336
Глобальные переменные	341
API математической библиотеки	341
Тригонометрические функции	341
Функции для работы с системами координат	342
Функции для работы с векторами	345
Функции для работы с матрицами	353
Функции для работы с двумерными и трехмерными параметрическими прямыми	365
Функции для работы с трехмерными плоскостями	369
Функции для работы с кватернионами	373
Функции для работы с числами с фиксированной точкой	382
Функции для решения систем уравнений	387
Работа математического сопроцессора	390

Архитектура сопроцессора	390
Стек сопроцессора	391
Набор команд сопроцессора	393
Классический формат команд	396
Формат работы с памятью	396
Формат работы с регистрами	397
Формат работы с регистрами и снятие со стека	397
Примеры команд сопроцессора	398
Замечания по использованию математической библиотеки	406
Замечания об оптимизации	407
Резюме	408
<b>Глава 6. Введение в трехмерную графику</b>	<b>409</b>
Философия трехмерного игрового процессора	410
Структура трехмерного игрового процессора	410
Трехмерный процессор	411
Игровой процессор	411
Система ввода и работы в сети	412
Анимация	412
Система навигации и обнаружения столкновений	418
Физический процессор	419
Система <b>искусственного</b> интеллекта	420
База данных трехмерных моделей и изображений	421
Трехмерные системы координат	423
Координаты модели (локальные координаты)	424
Мировые координаты	427
Координаты камеры	430
Дополнительные вопросы	439
Отбраковка скрытых объектов и поверхностей	441
Аксонметрические координаты	447
Экранные координаты	458
Базовые трехмерные структуры данных	467
Представление трехмерных многоугольников	468
Определение многоугольников	470
Определение объектов	476
Представление миров	481
Инструментарий трехмерного моделирования	483
Анимационные данные	484
Загрузка внешних данных	484
PLG	485
NFF	488
3D Studio	492
COB-файлы Caligari	499
.X-файлы Microsoft DirectX	501
Краткое резюме	501
Основы преобразований твердых тел и анимации	502
Трехмерное перемещение	502
Трехмерное вращение	503
Морфинг	506

Обзор конвейера визуализации	507
Типы трехмерных игровых процессоров	508
Космические процессоры	508
Наземные процессоры	510
FSP-процессоры	511
Трассировка лучей и воксельные процессоры	512
Гибридные процессоры	513
Сборка игрового процессора	514
Резюме	514
<b>Глава 7. Визуализация трехмерных каркасных объектов</b>	<b>515</b>
Общая архитектура каркасного игрового процессора	516
Структуры данных и трехмерный игровой конвейер	517
Основной список многоугольников	521
Новые программные модули	524
Создание простого загрузчика файлов трехмерными моделями	524
Загрузчик файлов в формате .PLG/X	528
Разработка трехмерного игрового конвейера	536
Функции преобразования общего вида	537
Преобразование из локальной системы координат в мировую	545
Эйлерова модель камеры	550
Модель UVN камеры	554
Преобразование мировых координат в координаты камеры	569
Отбраковка объектов	574
Удаление обратных поверхностей	579
Аксонметрическое преобразование	583
Преобразование аксонометрических координат в экранные	589
Аксонметрическое преобразование в экранные координаты	596
Визуализация трехмерного игрового мира	600
Трехмерный игровой конвейер	600
Трехмерные демонстрационные программы	605
Вывод отдельного треугольника	605
Вывод трехмерного каркаса куба	608
Вывод трехмерного каркаса куба с удалением обратных поверхностей	612
Трехмерные танки	614
Трехмерные танки и перемещаемая камера	618
Перемещение по зоне боевых действий	620
Резюме	626
<b>Часть III. Основы трехмерной визуализации</b>	<b>627</b>
<b>Глава 8. Основы моделирования освещения и поверхностей тел</b>	<b>629</b>
Основные модели освещения в компьютерной графике	630
Цветовые модели и материалы	633
Виды освещения	643
Освещение и растеризация треугольников	652
Подготовка к моделированию освещения	657
Моделирование материалов	659
Определение источников освещения	664

Затенение в реальном мире	671
16-битовое затенение	671
8-битовое затенение	672
Сложная RGB-модель для 8-битового режима	672
Упрощенная модель интенсивности освещения для 8-битовых режимов	678
Постоянное затенение	684
Плоское затенение	687
Плоское затенение в 8-битовом режиме	707
Затенение по Гуро	710
Затенение по Фонгу	713
Сортировка по глубине и алгоритм художника	714
Работа с новыми форматами моделей	721
Класс-анализатор	721
Формат .ASC 3D Studio MAX	728
Текстовый формат .COB trueSpace	732
Первое знакомство с бинарным форматом Quake II .MD2	743
Обзор программных инструментов для создания трехмерных моделей	744
Резюме	748
<b>Глава 9. Интерполяционные методы затенения и аффинное отображение текстур</b>	<b>749</b>
Особенности нового трехмерного процессора	750
Обновление и разработка структур трехмерных данных	751
Новые директивы #define	752
Добавление математических структур	755
Вспомогательные макросы	756
Новые возможности представления трехмерных каркасов	757
Обновление структур объектов и списка визуализации	766
Обзор списка функций и их прототипов	772
Новые версии загрузчиков	780
Обновление загрузчика .PLG/PLX-файлов	781
Обновление загрузчика файлов в формате .ASC	797
Обновление загрузчика файлов в формате Caligari .COB	798
Обзор растеризации многоугольников	805
Растеризация треугольников	805
Соглашение о заполнении	810
Отсечение графического изображения	814
Новые функции, связанные с обработкой треугольников	815
Оптимизация	821
Реализация затенения по Гуро	823
Затенение по Гуро без освещения	825
Добавление освещения вершин в функцию, выполняющую затенение по Гуро	837
Основы теории дискретизации	849
Дискретизация в одном измерении	850
Билинейная интерполяция	853
Интерполяция координат и $u$ и $v$	854
Реализация аффинного отображения текстуры	858
Обновление процессора освещения и растеризации для работы с текстурами	861
Добавление освещения для визуализации текстуры в 16-битовом режиме	862

Вопросы оптимизации для 8- и 16-битовых режимов	869
Таблицы соответствия	870
Связность вершин каркаса	870
Кэширование математических результатов	871
Использование возможностей SIMD	872
Итоговые демонстрационные программы	872
Raiders 3D II	873
Резюме	877
<b>Глава 10. Отсечение в трехмерном пространстве</b>	<b>879</b>
Общее представление об отсечении	880
Отсечение в пространстве объекта	880
Отсечение в пространстве изображений	884
Теоретические основы алгоритмов отсечения	884
Основы отсечения	886
Отсечение по Кохену-Сазерленду	891
Отсечение по Сайрусу-Беку и Лянгу-Барскому	893
Отсечение по Вейлеру-Азертону	897
Дальнейшее изучение отсечения	900
Практическое отсечение по границам области обзора	900
Конвейер обработки геометрии и структуры данных	902
Добавление отсечения в игровой процессор	903
Игры с ландшафтами	935
Функция, генерирующая ландшафты	936
Генерация данных ландшафта	949
Демонстрационная программа	949
Резюме	955
<b>Глава 11. Организация буфера глубины и видимость</b>	<b>957</b>
Буфер глубины и определение видимости	958
Основные принципы работы с Z-буфером	961
Проблемы, связанные с Z-буфером	963
Примеры Z-буфера	964
Метод уравнивания плоскости	967
Интерполяция координаты $z$	968
Проблемы, связанные с Z-буфером, и $1/Z$ -буферизация	971
Пример интерполяции по $Z$ и $1/Z$	972
Создание системы Z-буферизации	975
Добавление поддержки Z-буфера в функцию растеризации	979
Возможные оптимизации Z-буфера	998
Сокращение объема используемой памяти	998
Менее частая очистка Z-буфера	998
Смешанная Z-буферизация	1001
Сложности при работе с Z-буфером	1001
Демонстрационные программы, использующие Z-буферизацию	1002
Первая демонстрационная программа: визуализация Z-буфера	1002
Вторая демонстрационная программа: водный мотоцикл	1004
Резюме	1011

<b>Часть IV. Секреты трехмерной визуализации</b>	<b>1013</b>
<b>Глава 12. Методы сложного текстурирования</b>	<b>1015</b>
Текстурирование — вторая волна	1016
Структуры данных заголовочного файла	1017
Построение нового базового растеризатора	1026
Работа с числами с фиксированной точкой	1027
Новые растеризаторы без Z-буферизации	1027
Новые растеризаторы с Z-буфером	1032
Текстурирование с затенением по Гуро	1034
Прозрачность и альфа-смешивание	1043
Использование таблиц поиска для альфа-смешивания	1043
Поддержка альфа-смешивания на уровне объектов	1057
Добавление поддержки альфа-смешивания в генератор ландшафта	1065
Текстурирование с корректной перспективой и 1/z-буферизация	106S
Математические основы отображения текстур с корректной перспективой	1069
Добавление 1/z-буферизации в растеризаторы	1078
Реализация отображения с точной перспективой	1086
Отображение текстуры с линейно-кусочной перспективой	1090
Квадратичные аппроксимации для текстурирования с перспективой	1097
Оптимизация текстурирования с помощью гибридных подходов	1101
Билинейная фильтрация текстуры	1103
Множественное отображение и трилинейная фильтрация текстур	1108
Введение в Фурье-анализ	1111
Создание цепочки множественного отображения	1114
Выбор метода множественного отображения	1126
Трилинейная фильтрация	1132
Многопроходная визуализация и Текстурирование	1133
Все в одном вызове	1134
Новый контекст визуализации	1135
Настройка контекста визуализации	1138
Вызов функции визуализации	1140
Резюме	1141
<b>Глава 13. Алгоритмы разбиения пространства и определения видимости</b>	<b>1143</b>
Новый модуль игрового процессора	1144
Разбиение пространства и определение видимости	1144
Двоичное разбиение пространства	1148
Двоичное разбиение пространства плоскостями, параллельными осям	1150
Разбиение пространства произвольными плоскостями	1151
Разбиение с помощью плоскостей, определяемых многоугольниками	1152
Отображение/посещение каждого узла BSP-дерева	1156
Функции и структуры данных BSP-деревьев	1159
Создание BSP-дерева	1161
Стратегии расщепления	1165
Обход и отображение BSP-дерева	1177
Интеграция BSP-дерева в графический конвейер	1191
Редактор уровня	1193
Недостатки BSP	1205

Стратегии с нулевым перерисовыванием на основе BSP	1205
Использование BSP-деревьев для отбраковки	1208
Использование BSP-деревьев для выявления столкновений	1219
Интеграция BSP-деревьев в стандартную визуализацию	1219
Потенциально видимые множества	1225
Использование потенциально видимого множества	1228
Методы представления потенциально видимого множества	1230
Более точное вычисление PVS	1231
Порталы	1234
Ограничивающие иерархические объемы и октадеревья	1237
Использование BVH-деревя	1240
Производительность	1241
Стратегии выбора	1242
Октадеревья	1255
Отбор с учетом препятствий	1257
Перекрытые пространства	1258
Выбор преград	1260
Гибридный метод выбора преград	1260
Резюме	1261
<b>Глава 14. Освещение и тени</b>	<b>1263</b>
Новый модуль игрового процессора	1264
Введение и план игры	1264
Упрощенная физика теней	1264
Движение фотонов и вычислительная интенсивность	1265
Имитация теней с помощью проецирования изображений и макетов	1269
Растеризация с поддержкой прозрачности	1271
Новый библиотечный модуль	1275
Простые тени	1276
Масштабирование теней	1279
Отслеживание источника света	1283
Последние замечания об имитации теней	1289
Отображение тени на плоскую сетку	1290
Вычисление векторных преобразований для проекции	1290
Оптимизация плоских теней	1295
Введение в отображение освещения и кэширование поверхности	1296
Кэширование поверхности	1299
Генерация карт освещения	1300
Реализация отображения освещения	1301
Отображение темноты	1305
Получение специальных эффектов с помощью карт освещения	1307
Оптимизация кода отображения освещения	1308
Резюме	1308
<b>Часть V. Анимация, физическое моделирование и оптимизация</b>	<b>1309</b>
<b>Глава 15. Анимация, движение и обнаружение столкновений</b>	<b>1311</b>
Новый модуль игрового процессора	1312
Введение в трехмерную анимацию	1312

Формат Quake II .MD2	1312
Заголовок .MD2	1315
Загрузка .MD2-файлов Quake II	1325
Анимация файлов .MD2	1336
Простая анимация без участия персонажа	1349
Вращательное и поступательное движение	1349
Сложное параметрическое и криволинейное движение	1350
Использование сценариев движения	1352
Обнаружение трехмерных столкновений	1354
Ограничивающие сферы и цилиндры	1355
Использование структур данных для ускорения обнаружения столкновений	1356
Техника следования по поверхности	1357
Резюме	1358
<b>Глава 16. Технологии оптимизации</b>	1359
Введение в оптимизацию	1360
Профилирование кода программы	1360
Профилирование с помощью Visual C++	1361
Анализ результатов профилирования	1364
Оптимизация с помощью VTune	1365
Использование компилятора Intel C++	1372
Загрузка оптимизирующего компилятора Intel	1373
Использование компилятора Intel	1374
Использование дополнительных возможностей компилятора	1375
Выбор компилятора для исходных файлов	1376
Стратегии оптимизации	1377
Введение в SIMD	1377
Базовая архитектура SIMD	1379
Работа с SIMD в реальном мире	1379
Класс трехмерных векторов с поддержкой SIMD	1392
Некоторые оптимизационные приемы	1400
Прием 1. Избавление от _ftol()	1400
Прием 2. Задание управляющего слова FPU	1400
Прием 3. Быстрое обнуление значений с плавающей точкой	1401
Прием 4. Быстрое извлечение квадратного корня	1402
Прием 5. Линейно-кусочный арктангенс	1402
Прием 6. Увеличение указателя	1402
Прием 7. Вынесение if из циклов	1403
Прием 8. Ветвление конвейера	1404
Прием 9. Выравнивание данных	1404
Прием 10. Все короткие функции нужно сделать встраиваемыми	1404
Резюме	1405
<b>Предметный указатель</b>	1407

## Предисловие

Для меня большая честь написать предисловие к этой важной работе, с базового уровня обучающей навыкам программирования, необходимым для создания трехмерных видеоигр нового поколения. Существует не так уж много книг, которые учат создавать трехмерную виртуальную машину, работающую в реальном времени, начиная с простейшего вывода пикселей на экран. Как все же далеко продвинулись технологии игр с тех пор, как были созданы первые простейшие игры для Atari. В те времена игры, кажущиеся примитивными сейчас, потрясали воображение.

Сегодня очевидно, что первые игры с технической точки зрения не были компьютерными играми. На самом деле они были всего лишь забавными генераторами сигналов и конечными автоматами, использующими счетчики и регистры сдвигов, основанные на простой булевой логике. Они были не чем иным, как совокупностью логических схем среднего уровня интеграции. Моя первая игра *Computer Space* была создана в 1970г., за четыре года до появления микропроцессора Intel 4004 и за шесть лет до появления процессора 8080. Мне так хотелось иметь микропроцессор, чтобы делать настоящие игры! Даже по тем временам типичная рабочая частота была слишком мала, чтобы производить расчеты в реальном времени. Впервые мы использовали микропроцессор в игре *Asteroids*. И даже тогда для ускорения работы программы использовалась масса дополнительного оборудования, поскольку микропроцессор не мог справиться со всеми поставленными задачами.

Сегодня мы движемся к возможности создания сцен фотографического качества (если еще не пришли к ней). Процесс создания таких сцен в реальном времени приносит истинное наслаждение. Реализм и возможность создания любых миров, окружающей среды и персонажей, достигаемые благодаря современному программному обеспечению и оборудованию, дают создателю игр потрясающую власть. Эти возможности позволяют сжимать время, увеличивать богатство и реализовать новые, доселе немыслимые проекты.

Андре Ламот не только глубоко понимает технологии, использующиеся при разработке игр, но и имеет экстраординарное "чутье" игры. Мой многолетний опыт показывает, что нередко разработчики являются великими знатоками своего дела, но им не хватает чувства времени или напряжения, которые так необходимы для увлекательной игры. Другие хорошо чувствуют игру, но они слабые программисты. Андре сочетает в себе качества увлеченного игрока и профессионального программиста, и это видно во всех написанных им книгах.

В последнем нашем совместном проекте на меня произвел впечатление не только его профессионализм, но и знание истории и близкое знакомство даже с малоизвестными самыми первыми играми (до этого я считал, что о них помню только я). Не могу не упомянуть о том, что он написал для меня игру всего за 19 дней! Легко помнить о хитах, иное дело — знать неудачные игровые программы (надо сказать правду — Atari сделала несколько откровенно слабых проектов). А мы, как известно, сделали много игр, которые стали, не побоюсь этого слова, классическими.

Надеюсь, что вам понравится эта книга. Используйте ее как стартовую площадку для создания новых великих игр, способных осчастливить человечество.

Нолан Башнелл  
Учредитель Atari, Inc.

## Об авторе

Андре Ламот связан с компьютерной индустрией и информационными технологиями более четверти столетия. Он является обладателем многочисленных дипломов в области математики, компьютерных наук и электротехники. Он — один из тех редких людей, которые действительно делают серьезную работу в NASA. Уже с молодых лет Андре занимался консультированием многочисленных компаний Силиконовой Долины, где он на практике осваивал бизнес и использовал свои разносторонние знания в таких областях, как телекоммуникации, виртуальная реальность, робототехника, разработка компиляторов, трехмерные виртуальные машины, искусственный интеллект, и в других областях информационных технологий и проектирования.

Его компания Xtreme Games LLC была одной из первых (и последних) небольших компаний "со своей душой". Позднее он основал Xtreme Games Developer Conference (XGDC) в качестве дешевой альтернативы GDC (Games Developer Conference — конференция разработчиков игр).

Ламот работал над рядом известных проектов, включая eGamezone Networks, который представляет собой интерактивную систему распространения игр — честную, с юмором и без рекламы. Андре основал новую компанию, Nurve Networks LLC, которая занялась разработкой переносных игровых видеосистем для пользователей с высокими требованиями к играм. Наконец, он является редактором крупнейшего в мире периодического издания по разработке игр.

В свободное время он увлекается всем экстремальным — от тяжелой атлетики и мотоциклов до боевых единоборств. Одно время он даже интенсивно тренировался в составе команды Shamrock Submission Fighting Team под руководством именитых Крэйзи Боба Кука (Crazy Bob Cook), Фрэнка Шэмрока (Frank Shamrock) и Жавье Мендеса (Javier Mendez). Я думаю, вам не захочется вступать с ним в дискуссию по поводу DirectX или OpenGL — прав он или нет, последнее слово все равно останется за ним!

## О техническом редакторе

Дэвид Фрэнсон (David Franson) профессионально работает в области сетевых технологий, программирования, двумерной и трехмерной компьютерной графики с 1990г. В 2000г. он ушел с должности директора информационных систем одной из крупнейших юридических фирм Нью-Йорка, чтобы полностью посвятить себя разработке игр. Он также является автором книги *2D Artwork and 3D Modeling for Game Artists*, изданной Premier Press, и в настоящее время он работает над книгой, посвященной *Xbox Hackz and Modz*.

*Посвящаю эту книгу всем честным людям — даже если это путь одиночества, кто-то должен идти этой дорогой. Ваш труд не останется незамеченным...*

## Благодарности

Прежде всего мне хотелось бы выразить свою благодарность сотрудникам Sams Publishing. Я позволю себе заметить, что в этом издательстве сложился довольно стандартный подход к изданию книг, и серия моих книг **выводит** их из равновесия. Мы должны действительно поаплодировать умению этих людей отойти от корпоративного стиля и предоставить авторам **возможность** создать нечто особенное. Помните об этом, когда будете читать их имена.

Я бы хотел выразить свою признательность издателю Майклу Стефенсу (Michael Stephens), редактору по работе с авторами Киму Спилкеру (Kim Spilker), редактору по подготовке текста Марку Ренфроу (Mark Renfrow) и, конечно, редактору проекта, который отвечает за то, чтобы весь механизм издания книги работал слаженно, — Джорджу Недэффу (George Nedeff). Нельзя не упомянуть об основных "двигателях" этого механизма — редакторе Сете Керни (Seth Kerney) и техническом редакторе Дэвиде Фрэнсоне (David Franson) (Дэвид также предоставил некоторые трехмерные модели, включая Jet Ski, а также многие текстуры, использованные в демонстрационных программах).

И наконец, поскольку сегодня никакая книга не выглядит завершенной без компакт-диска, я выражаю признательность мультимедиа-разработчику Дэну Шерфу (Dan Scherf), а также Эрике Миллен (Erika Millen), благодаря которой был создан подробный и исчерпывающий предметный указатель книги, что важно, когда вам нужно что-то найти в тексте.

Есть еще один особенный человек, внесший значительный вклад на начальном этапе создания книги. Его имя Анжела Козловски (Angela Kozlowski). Она работала со мной на начальном этапе создания книги и по-настоящему помогла мне определить ее общую структуру. Она помогла всем понять, как важна и как необычна эта книга. Она не смогла увидеть завершающий этап создания книги, поскольку перешла на другую работу, но, я уверен, она бы этого хотела,

Следует также упомянуть компании и людей, которые дали, одолжили или каким-то иным способом обеспечили мне доступ к программному обеспечению и оборудованию. Первый из них — это Дэвид Хэксон (David Hackson) из Intel Corporation, который дал мне последние версии компиляторов C++ и Fortran, а также VTune. Кристин Гарднер (Kristine Gardner) из Caligari Corporation дала мне все когда-либо созданные версии trueSpace. Моя давняя знакомая Стэси Цурусаки (Stacey Tsurusaki) из Microsoft предоставила мне возможность "взглянуть" на Microsoft изнутри. И конечно, были компании, предоставившие мне пробные версии своих продуктов или давшие возможность тем или иным способом записать их программы на компакт-диски. — это JASC Inc. (Paint Shop Pro) и Sonic Foundry (Sound Forge). Мэри Элис Краецки (Mary Alice Krayecki) из Right Hemisphere прислала мне копии Deep Exploration и Deep Paint UV.

Мне хотелось бы также перечислить людей, которые поддерживали со мной дружеские отношения, пока я работал над книгой. Я знаю, что я "интенсивная" личность, но именно это делает меня столь забавным на вечеринках!

В любом **случае**, мне хотелось бы выразить признательность Майку Перону (Mike Perone) за **помощь** в приобретении программного обеспечения, а также при работе с сетевыми вопросами.

Марку Беллу (Mark Bell) за выслушивание моих жалоб — только бизнесмен способен понять тот ад, через который мы прошли, — спасибо, Марк.

Селламу **Исмаилу** (Sellam Ismail) из Vintage Computer Festival за массу бесподобного ретро-материала.

Джону Ромеро (John Romero) за его частое общение со мной и за то, что в нашем бизнесе есть хоть **кто-нибудь**, кого можно было бы назвать забавным!

**Нолану Башнеллу** (Nolan Bushnell) за то, что он принял меня в uWink Inc. и нашел время для разговора со мной о видеоиграх. Ведь не каждый день вы проводите время в обществе **человека**, создавшего Atari! И спасибо за написание предисловия!

Алексу Варанесу (Alex Varanese), моему новому ученику, за его готовность следовать моим постоянным проповедям о совершенствовании...

И **наконец**, **людям**, которые **по-настоящему** испытали на себе остроту моей агрессивной и требовательной личности — маме, отцу и моей терпеливой подруге Аните.

## Ждем ваших отзывов!

Вы, уважаемый читатель, и есть **главный** критик и комментатор этой книги. Мы **ценим** ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые **вам** хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш **Web-сервер** и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для **вас**.

Посылая письмо или **сообщение**, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

**E-mail:** [info@williamspublishing.com](mailto:info@williamspublishing.com)  
**WWW:** <http://www.wiUiamspublishing.com>

Информация для писем:

из России: 115419, Москва, а/я 783  
из Украины: 03150, Киев, а/я 152

# ВВЕДЕНИЕ

## Принципы и практика программирования игр

Давным-давно, далеко-далеко отсюда я написал книгу под названием *Программирование игр для Windows. Советы профессионала*. Для меня это была возможность сделать то, о чем я давно мечтал, — создать книгу, которая учит делать игры. Прошло несколько лет, я стал немного старше, немного мудрее и узнал массу новых приемов. Вы можете обойтись и без чтения упомянутой книги, однако я предупреждаю — книга, которую вы держите в руках, намного более сложная. Она сфокусирована на трехмерной графике и предполагает определенную подготовку у читателя.

Эта книга продолжает разговор с того места, на котором он был прерван в первой книге, и подробно рассматривает **принципы** трехмерной графики. Я собираюсь рассмотреть все основные темы программирования трехмерных игр — все, которые мне удастся "уложить" в отведенные рамки объема книги и сроков ее написания.

Я не рассчитываю, что вы являетесь асом программирования или что вы **знаете**, как делать игры. Однако эта книга — определенно не для новичков, она ориентирована на разработчиков игр и программистов среднего или высокого уровня. Если вы не знаете, как программировать на C++, — вам здесь нечего делать. Так что, **новички**, хорошенько подумайте — не стоит ли потратить деньги на хороший учебник по C++ (я бы предложил что-нибудь, написанное Стефеном Прата (Stephen Prata) или Робертом Лафором (Robert Lafore), по моему мнению, это лучшие авторы по данной теме).

Сейчас, вероятно, самое горячее время в истории игрового бизнеса. Сегодня в руках программистов оказались технологии создания игр, которые выглядят совершенно **реалистично**. Представьте, каким будет следующее поколение игр (а вы все еще думаете, что PlayStation 2, Xbox, и GameCube — это **круто?**). Однако все эти технологии нетривиальны и сложны для понимания — они возникли в результате упорной работы многих умных людей.

Уровень сложности программирования игр сегодня **существенно** вырос. Но если вы это читаете, то, вероятно, вы один из тех, кто любит бросать вызов судьбе, не так ли? Ну что же, тогда вы обратились по адресу, ведь если вы одолеете эту книгу, то сможете самостоятельно создать игру полностью трехмерную, с текстурами с полной проработкой освещения и программной **растеризацией**. Кроме того, вы изучите базовые принципы трехмерной графики и будете **лучше** понимать и уметь использовать **оборудова-**

ние для трехмерной графики как **сегодняшнее**, так и то, которое появится в ближайшем **будущем**.

## Что вы сможете изучить

Эта книга содержит огромный объем информации! Если заполнить ею ваш мозг, у вас начнется утечка нейронов! :-) А если серьезно, то эта книга излагает все сведения, необходимые для создания игр для платформы Windows 9x/2000, включая следующие разделы.

- Принципы создания игрового **процессора** из первой книги.
- Программирование 32 и DirectX.
- Высшая математика, включая кватернионы.
- Двумерная и трехмерная графика и алгоритмы.
- Трехмерные проекции и работа с камерой.
- Визуализация каркасов и сплошных тел.
- Работа со светом и текстурами.
- Сложные алгоритмы визуализации.
- Методы трехмерной анимации.

И многое **другое...**

Первое, о чем меня можно спросить, — рассматриваются ли в этой книге вопросы использования оборудования для трехмерной графики, или показывает ли она, как реализовать трехмерную программную растеризацию? Ответ — книга в первую очередь посвящена написанию программ. Только дилетанты полагаются на аппаратное обеспечение. Настоящий программист игр может написать трехмерный процессор с нуля даже под артобстрелом, когда лопаются барабанные перепонки. При этом он сможет использовать и **оборудование** — если оно есть. Поэтому в данной книге мы сосредоточимся на **программировании** трехмерных игр. Обладая этими знаниями, можно изучить любой API для **трехмерной** графики в течение пары недель.

Моя философия такова: если вы знаете, как написать программу отображения текстуры или систему визуализации сцены, то это дает вам значительно больше, чем знания оборудования. Кроме того, потребуется какое-то время для того, чтобы в каждом компьютере было установлено хорошее оборудование для трехмерной графики. Неправильно считать, что оно установлено на любом компьютере, и вычеркивать из своего целевого рынка компьютеры только потому, что у них нет такого оборудования. (Средний Запад и Европа — великолепные примеры регионов, в которых нет последних новинок технологий). А если к тому же у вас нет миллионов долларов для завоевания игрового рынка, то вам прямая дорога на рынок программных игр, которые рассчитаны на компьютеры без ускорителей **3D-графики**.

Наконец, я **уверен**, что вы испытываете некоторое беспокойство насчет всей этой затеи с "Windows-DirectX". Дело в том, что при правильном подходе программирование для Windows — занятие очень простое и забавное, которое снимает многие проблемы, которые имеют место в случае с DOS32. Не следует думать о программировании под Windows как о проблеме — думайте об этом как о еще одной возможности посвятить свое время работе над игровым кодом, а не над такими вещами, как графический интерфейс пользователя, система ввода-вывода и графические драйверы. Поверьте мне, вы будете работать круглосуточно, если захотите написать **графи-**

ческие драйверы для всех двумерных или трехмерных ускорителей, выходящих на рынок. Кроме того, есть еще звуковые карты, джойстики и много чего еще...

## Что необходимо знать

Эта книга предполагает, что вы умеете программировать, и программировать очень хорошо. Вы здесь просто потеряетесь, если не сможете написать код на С или не будете отлично знать, как использовать компилятор. Кроме того, в книге используется С++, чего уже достаточно, чтобы С-кодировщик чувствовал себя немного неуютно. Я сообщу вам, когда буду делать что-то сверхъестественное. В приложениях содержится дополнительный учебный материал по С++, поэтому воспользуйтесь им, если вам необходим начальный курс. Впрочем, С++ в основном нужен только при использовании DirectX.

Тем не менее, я решил, что в этой книге буду использовать С++ немного шире, поскольку в программировании игр есть довольно много объектно-ориентированных вещей, и в таких случаях использовать структуры языка С — это просто кошунство. Подвожу итог: если вы умеете писать программы на С, вы все поймете. Если умеете писать их и на С, и на С++ — вам вообще не о чем беспокоиться.

Каждый знает, что компьютерная программа — это просто логика и математика. Трехмерные видеоигры делают упор именно на математику! 3D-графика — это только математика. К счастью для нас, это красивая математика. (Да, математика может быть красивой.) Дело в том, что все, что вам нужно знать, — это алгебра и геометрия. Векторам, матрицам и дифференциальному исчислению я обучу вас по ходу дела. Если вы умеете складывать, вычитать, умножать и делить, вы сможете понять большую часть материала, даже если не сможете повторить ход рассуждений. Но если при этом вы сможете использовать представленный код — то мы добились нашей конечной цели.

Фактически это все, что нужно знать. Конечно, вам лучше позвонить своим друзьям и сказать, что они не смогут видиться с вами около двух лет, поскольку вы будете немного заняты. Но вы только подумайте обо всех тех новых фильмах, которые сможете посмотреть после окончания обучения!

## Структура книги

Эта книга состоит из шести основных частей.

- Часть I, "Введение в программирование трехмерных игр". В этой части излагаются основы программирования игр, Windows, DirectX, здесь мы также построим виртуальный компьютерный интерфейс для разработки демонстрационных программ.
- Часть II, "Трехмерная математика и преобразования". В этом разделе описываются используемые в дальнейшем математические концепции, а также создается математическая библиотека, используемая в данной книге. Последние главы раздела посвящены трехмерной графике, структурам данных, камерам и каркасной визуализации.
- Часть III, "Основы трехмерной визуализации". Этот раздел охватывает такие темы, как освещение, основы наложения теней и удаление невидимых поверхностей. Рассматривается также полное 3D-отсечение.
- Часть IV, "Секреты трехмерной визуализации". Рассматриваются наложение текстур, сложное освещение, сложные тени, а также алгоритмы пространственного разделения, такие как BSP-деревья, порталы и другие.

- **Часть V**, "Анимация, физическое моделирование и оптимизация". В этой части описываются **анимация**, движение объектов, обнаружение столкновений и простое физическое моделирование. Рассматривается также иерархическое моделирование вместе с загрузкой больших игровых миров. Здесь также описаны многочисленные методы оптимизации.

Шестая часть представляет собой находящиеся на прилагаемом компакт-диске приложения.

## Установка содержимого CD-ROM

CD-ROM содержит все исходные **тексты**, исполняемые модули, примеры программ, библиотеку графических объектов, программы трехмерного **моделирования**, звуковые эффекты, а также технические статьи, которые дополняют книгу. Структура каталогов диска представлена ниже.

T3DIIGAME\

SOURCE\

T3DIICHAP01\

T3DIICHAP02\

T3DIICHAP16\

TOOLS\

GAMES\

MEDIA\

BITMAPS\

3DMODELS\

SOUND\

DIRECTX\

ARTICLES\

В каждом каталоге имеются различные необходимые материалы. Далее следует их более **подробное описание**.

- **T3DGAMEII** — корневой каталог, в котором находятся все остальные каталоги. Прочитайте файл **README.TXT**, в котором описаны все последние изменения.
- **SOURCE** — содержит все каталоги с исходными кодами книги в порядке **следования** глав. Просто перетяните мышью весь каталог **SOURCE\** на ваш жесткий диск и работайте с ним оттуда.
- **MEDIA** — содержит набор двумерных изображений, трехмерных моделей и звуков, которые вы можете бесплатно использовать в своих играх.
- **DIRECTX** — содержит последнюю версию DirectX SDK.
- **GAMES** — содержит несколько двумерных и трехмерных условно-бесплатных игр, которые, по моему **мнению**, просто превосходны!
- **ARTICLES** — содержит статьи, написанные для вашего развития различными специалистами в области программирования игр.

- **TOOLS** — содержит различные полезные приложения и инструменты. Большинство из них можно загрузить из Internet, однако они довольно большие, и данные файлы позволят вам сэкономить на времени загрузки.

Для компакт-диска нет единой программы установки, поскольку здесь находится много различных программ и данных. Оставляю установку на ваше усмотрение. В большинстве случаев можно просто скопировать каталог **SOURCE\** на жесткий диск и работать с ним оттуда. Что касается других программ и данных, установите те из них, которые вам нужны.

## Установка DirectX

Пожалуй, единственная часть компакт-диска, которую нужно установить обязательно, — это DirectX SDK и файлы времени выполнения. Программа установки находится в каталоге **DIRECTX\** вместе с файлом **README.TXT**, в котором указаны все последние изменения.

НА ЗАМЕТКУ

Для работы с демонстрационными программами или исходными файлами этой книги вам необходимо установить DirectX 8.1 SDK или более позднюю версию (на компакт-диске находится DirectX 9.0), поэтому если вы точно не знаете, какая версия файлов у вас установлена, — запустите программу установки, и она сообщит вам об этом.

## Компиляция программ

Я написал программы для этой книги с помощью Microsoft Visual C++ 6.0. Тем не менее, в большинстве случаев программы будут работать с любым компилятором, совместимым с Win32. И все же я рекомендую Microsoft Visual C++ или .NET, поскольку с ними программы работают лучше всего.

Если вы никогда не работали с интегрированной средой разработки своего компилятора, то уделите время изучению компилятора — по крайней мере, научитесь компилировать консольное приложение "Hello World" или что-то подобное, прежде чем приступить к компиляции серьезных программ.

Для компиляции Win32 .EXE-модулей программ для Windows все, что нужно сделать, — это установить соответствующим образом опцию целевого программного проекта и приступить к компиляции. Однако для создания DirectX-программ необходимо включить в проект библиотеки импорта DirectX. Можно подумать, что достаточно просто добавить библиотеки DirectX в пути поиска компилятора, однако это не так. Поберегите свои нервные клетки и добавьте DirectX .LIB-файлы в проект или рабочую область вручную. .LIB-файлы можно найти в каталоге **LIB\**, в основном каталоге DirectX SDK, который вы установили. В этом случае при компоновке не будет никаких неприятных неожиданностей. В большинстве случаев вам необходимы следующие файлы.

- **DDRAW.LIB** — библиотека импорта DirectDraw
- **DINPUT.LIB** — библиотека импорта DirectInput
- **DINPUT8.LIB** — библиотека импорта DirectInput
- **DSOUND.LIB** — библиотека импорта DirectSound
- **WINMM.LIB** — Windows Multimedia Extensions

Я расскажу об этих файлах подробнее, когда мы начнем работать с ними. Сейчас же о них просто нужно помнить в случае сообщений компоновщика типа "unresolved

symbol". Я не хочу получать от новичков никаких электронных писем по этой теме и тем более не намерен отвечать на них!

Кроме .LIB-файлов DirectX, следует включить в путь поиска заголовочных файлов соответствующие .H-файлы. Убедитесь при этом, что каталоги DirectX SDK стоят первыми в пути поиска, поскольку многие C++-компиляторы содержат в себе старые версии DirectX, и в их каталоге заголовочных файлов могут находиться файлы старых версий DirectX, что неприемлемо. Надлежащее место — это каталог заголовочных файлов DirectX SDK.

Наконец, тем, кто пользуется продуктами Borland, следует убедиться, что у вас установлены Borland-версии .LIB-файлов DirectX. Их можно найти в каталоге BORLAND\DirectX SDK.



# ЧАСТЬ I

## Введение в программирование трехмерных игр

### В этой части...

#### Глава 1

Основы программирования трехмерных игр 29

#### Глава 2

Краткий курс Windows и DirectX 79

#### Глава 3

Виртуальный компьютер для программирования  
трехмерных игр 119



# ГЛАВА 1

## Основы программирования трехмерных игр

### В этой главе...

• Краткое введение	30
• Элементы двумерных и трехмерных игр	31
• Общие советы по программированию игр	36
• Использование инструментов	40
• Пример трехмерной игры: Raiders 3D	48

Для того чтобы разогреть ваш интерес, в этой главе мы рассмотрим некоторые общие темы программирования игр, такие как игровые циклы и различия между двумерными и трехмерными играми. В конце мы создадим небольшую трехмерную игру, чтобы правильно настроить компилятор и DirectX. Если вы уже прочитали мою первую книгу *Программирование игр для Windows. Советы профессионала*, то, наверное, можете бегло просмотреть эту главу и сосредоточиться на материале, изложенном в самом конце. Если же нет, вам определенно стоит прочитать ее — даже если вы игровой программист среднего или высокого уровня. Итак, вот содержание главы:

- краткое введение в программирование игр;
- элементы двумерных и трехмерных игр;
- общие советы по программированию игр;
- использование инструментов;
- пример игры для Windows: Raiders 3D.

## Краткое введение

Эта книга на самом деле представляет собой второй том серии (вероятно, трехтомной), посвященной программированию двумерных и трехмерных игр. В первом томе *Программирование игр для Windows. Советы профессионала* рассматриваются в первую очередь следующие темы.

- Программирование для Windows
- Интерфейс прикладного программирования (API) Win32
- Основы DirectX
- Искусственный интеллект
- Основы физического моделирования
- Звук и музыка
- Алгоритмы
- Программирование игр
- Двумерная растровая и векторная графика

Эта книга продолжает предыдущую. Тем не менее, я попытался написать ее так, что если вы не читали первую, то все равно сможете получить из данной книги массу информации по трехмерной графике реального времени и ее применению для создания трехмерных игр. Следовательно, идеальный читатель данной книги — это тот, кто прочел первую книгу серии и интересуется программированием трехмерных игр, либо тот, кто уже умеет делать двумерные игры и стремится освоить трехмерные методы с точки зрения написаний программ и алгоритмов.

Помня об этом, я собираюсь сосредоточиться в этой книге на трехмерной математике и графике, и в меньшей степени касаться всего, что относится к программированию игр. Я исхожу из того, что это вы уже умеете, — если нет, я в очередной раз советую вам прочитать *Программирование игр для Windows. Советы профессионала* (либо любую другую книгу по программированию игр) и посидеть за компьютером (до ощущения усталости), изучая Windows, DirectX и игровое программирование в целом.

С другой стороны, даже если вы не читали предыдущую книгу и ничего не знаете о программировании игр, вы все равно кое-что почерпнете для себя из этой книги — хотя бы потому, что здесь много иллюстраций. Мы собираемся заниматься от главы к главе созданием трехмерного игрового процессора, однако чтобы сэкономить время (и около 1500 страниц), мы начнем с базового процессора DirectX, разработанного в первой книге. Конечно, этот процессор использовал DirectX версий 7 и 8; сейчас DirectX изменился и в версии 8.0+ поддержка двумерных приложений стала сложнее, поскольку DirectDraw теперь объединен с Direct3D. Мы собираемся продолжить работу с интерфейсами DirectX 7 и 8, но компилировать наши приложения с DirectX 9.0.

### НА ЗАМЕТКУ

Это книга о программном обеспечении и алгоритмах, а не о DirectX. Поэтому я мог бы написать программу под DOS, и такой материал был бы вполне уместен. Нам необходимы только в меру сложная графика, система ввода/вывода и работа со звуком — для этих целей более чем достаточно DirectX 8.0+. Другими словами, если вы приверженец Linux, перенос игрового процессора под SDL пройдет без проблем!

Таким образом, если вы уже прочитали предыдущую книгу, игровой процессор и его функции уже знакомы вам. Если нет, его следует считать "черным ящиком" (код кото-

рого, впрочем, прилагается), поскольку в данной книге мы будем лишь добавлять трехмерный аспект к тому, что уже было сделано ранее.

Не беспокойтесь, в *следующей* главе мы полностью рассмотрим API и *структуру* двумерного игрового процессора, а также все функции, которые он выполняет. Я также покажу вам ряд демонстрационных программ, чтобы заинтересовать вас в использовании базового процессора DirectX, созданного в первой книге.

В этой книге я хочу попытаться представить максимально *общую*, "виртуальную" точку зрения на графическую систему. Хотя код будет основан на игровом процессоре из первой книги, главное в том, что этот игровой процессор не делает ничего, кроме настройки *графической* системы с двойной буферизацией со стандартной линейной адресацией.

Таким образом, если вы хотите перенести такой код на платформу *Mac* или *Linux*, он должен нормально заработать — после нескольких часов работы (максимум — до недели). Моя цель — научить вас работать с трехмерной графикой и математикой в целом. Так сложилось, что доминирующей вычислительной системой в настоящий момент является DirectX на платформе Windows, поэтому именно на ней я и построил низкоуровневую обработку. И мы посвятим наше время рассмотрению именно этих концепций верхнего уровня.

Мне уже приходилось писать о двумерных и трехмерных игровых процессорах, и у меня на жестком диске есть соответствующий материал. Однако когда я пишу книгу, я предпочитаю создавать новый игровой процессор, т.е. я пишу игровой процессор для книги, а не книгу для игрового процессора. В этой книге я не использую трехмерный игровой процессор повторно, а создаю его. Поэтому я не вполне уверен, что мне удастся эта книга! Мне даже интересно узнать, что же выйдет из этой затеи. Вы узнаете все необходимое для того, чтобы создать собственный игровой процессор Quake, но я могу по ходу дела переключиться на процессор для игр на открытом воздухе или какой-то еще — кто знает, куда меня занесет? Я исхожу из того, что работа над процессором гораздо полезнее, чем *код с примечаниями автора*.

Наконец, читатели моей первой книги могут *заметить*, что часть материала одинакова в обеих книгах. В самом деле, я не могу сразу начать с *3D-графики* без использования материала из предыдущей книги. Я не могу рассчитывать, что у каждого есть моя первая книга и не могу заставить купить ее. В любом случае, в первых нескольких главах *материал* будет несколько перекликаться с первой *книгой* — по крайней мере, в отношении DirectX, игрового процессора и Windows.

## Элементы двумерных и трехмерных игр

Для начала давайте посмотрим, чем видеоигра отличается от любого другого вида программ. *Видеоигры* — очень сложный вид программ, писать которые труднее всего. Конечно, написать что-нибудь типа MS Word труднее, чем игру типа *Asteroids*, но написать что-то вроде *Unreal*, *Quake Arena* или *Halo* труднее, чем любую *программу*, которую можно себе представить, включая военную программу управления вооружениями!

Это означает, что вам нужно изучить новый способ программирования, который более подходит для написания моделирующих приложений реального времени, чем программ, управляемых событиями, или последовательных логических программ. Видеоигра — это непрерывный цикл, в котором выполняется логический сценарий и рисуется картинка на экране — обычно с частотой не менее 30-60 кадров в секунду. Это напоминает фильм, но фильм, которым можно управлять.

Давайте начнем с рассмотрения упрощенной схемы игрового цикла (рис. 1.1). Вот краткое описание стадий цикла.

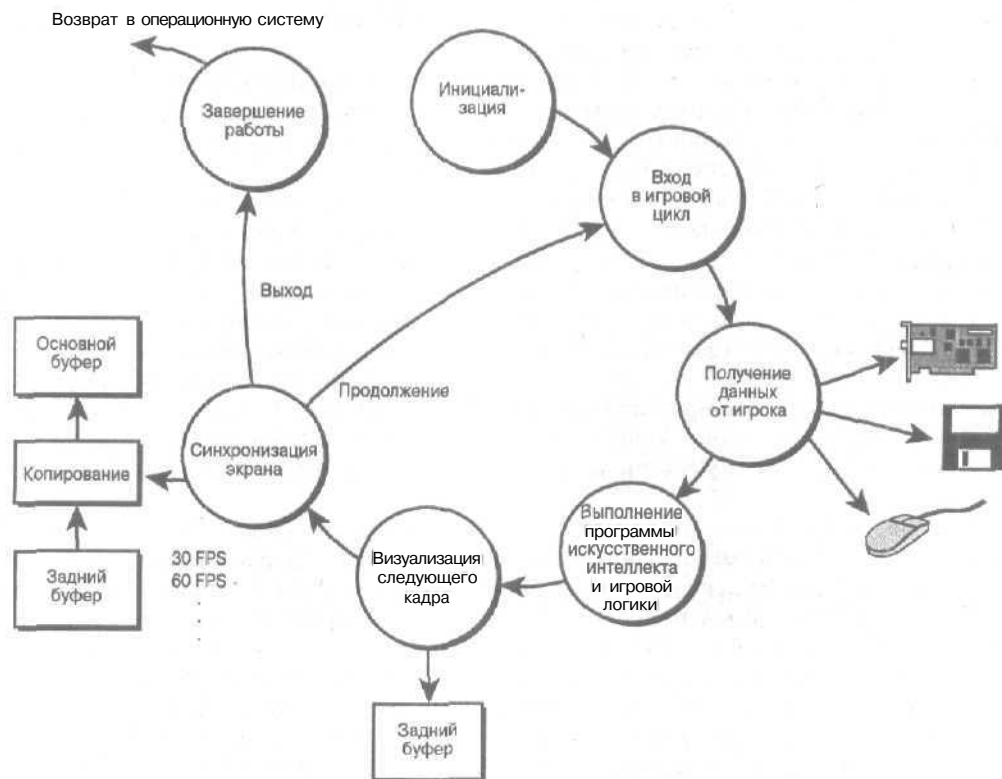


Рис. 1.1. Общая архитектура цикла игры

## Стадия 1. Инициализация

На этой стадии выполняются стандартные операции, характерные для любой программы, такие как выделение памяти, получение ресурсов, загрузка данных с диска и т.п.

## Стадия 2. Вход в игровой цикл

На этой стадии выполнение кода достигает входа в игровой цикл. Здесь начинается действие игры, и оно продолжается, пока пользователь не выйдет из основного цикла.

## Стадия 3. Получение данных от игрока

На этой стадии данные, введенные игроком, обрабатываются и/или заносятся в буфер для дальнейшего использования на стадии исполнения программы искусственного интеллекта и логики,

## Стадия 4. Выполнение программы искусственного интеллекта и игровой логики

Эта стадия содержит основную часть игрового кода. Исполняются программы искусственного интеллекта, физических систем, общей игровой логики. Результаты используются для формирования следующего кадра изображения.

## Стадия 5. Визуализация следующего кадра

На этой стадии данные, полученные от игрока, и результаты исполнения программы искусственного интеллекта и логики используются для формирования **следующего кадра** игрового изображения. Это изображение обычно формируется во внеэкранной буферной зоне. Его визуализацию невозможно увидеть. Затем происходит быстрое копирование изображения на экран, в результате чего возникает анимационный эффект. В случае трехмерного программного игрового процессора происходит визуализация тысяч (в некоторых случаях миллионов) многоугольников, формирующих игровую сцену. В случае трехмерного игрового процессора с аппаратным ускорением, основанного на **OpenGL** или **Direct3D**, значительная часть работы перекладывается на аппаратный ускоритель.

## Стадия 6. Синхронизация экрана

Скорость, с которой компьютер исполняет игровой код, зависит от уровня сложности игры. Например, если на экране находится **1000** движущихся объектов, нагрузка на центральный процессор будет значительно больше, чем в случае только с 10 объектами. Следовательно, частота кадров в игре будет меняться, что неприемлемо. Поэтому нужно обеспечить постоянную частоту кадров в игре с помощью функций для работы со временем и функций ожидания. В настоящее время **30fps** (кадров в секунду, frame per second) считается минимально допустимым значением, а **60fps** — **идеальным**. Частоты **выше 60fps** вряд ли оправданны, поскольку **мозг** с трудом обрабатывает информацию, поступающую с такой высокой скоростью.

### НА ЗАМЕТКУ

Хотя в ряде игр и достигается **идеальная** скорость 30-60 кадров в секунду, она может упасть с увеличением уровня сложности визуализации. Тем не менее, с помощью **контролируемых** по времени расчетов в разделах игровой логики, физики и искусственного интеллекта можно, по крайней мере, попытаться обеспечить согласование сцены во времени, т.е., например, при меньшей частоте обновления кадров объекты будут проходить большее расстояние между двумя кадрами.

## Стадия 7. Начало нового цикла

Эта стадия довольно проста — всего **лишь** возврат к началу игрового цикла и повторение **его** заново.

## Стадия 8. Завершение работы

Это конец игры, в том смысле, что пользователь выходит из основного раздела кода или игрового цикла и желает возвратиться в операционную систему. Однако, прежде чем сделать это, необходимо освободить все ресурсы и очистить систему так же, как это делается в случае исполнения любой программы.

Следует отметить, что приведенное объяснение немного упрощенное, однако оно верно отображает суть происходящего. В действительности игровой цикл в большинстве случаев является конечным автоматом, содержащим ряд состояний. Например, ниже Приведен более подробный код, иллюстрирующий, как может выглядеть игровой цикл

в реальном C/C++ коде.

```
// Состояния цикла игры
#define GAME_INIT // Инициализация игры
#define GAME_MENU // Режим меню
tfdefine GAME_START // Подготовка к запуску
tfdefine GAME_RUN // Работа игры запущена
#define GAME_RESTART // Подготовка к перезапуску
tfdefine GAME_EXIT // Выход из игры
```

```

// Глобальные переменные игры
int game_state = GAME_INIT; // Начинаем с этого состояния
int error = 0; // Используется для возврата
// сообщения об ошибке
// операционной системе

// Основная программа - Функция main
void main()
{
    // Реализация основного цикла игры
    while (game_state!=GAME_EXIT)
    {
        // Состояние игры
        switch(game_state)
        {
            case GAME_INIT: // Инициализация игры
            {
                // Выделение памяти и ресурсов
                Init();
                // Возврат в состояние меню
                game_state = GAME_MENU;
            } break;

            case GAME_MENU: // Режим меню
            {
                // Вызов основного меню и изменение
                // состояния игры
                game_state = Menu();
                // Примечание: здесь можно перейти в
                // состояние RUN
            } break;

            case GAME_START: // Игра готова к запуску
            {
                // Это состояние необязательно, но
                // обычно оно используется для настройки
                // состояния готовности. Здесь можно
                // заняться вспомогательными мероприятиями
                Setup_For_Run();

                // Переключаемся в состояние запуска
                game_state = GAME_RUN;
            } break;

            case GAME_RUN: // Игра запущена
            {
                // Здесь содержится полный цикл
                // логики игры

                // Очистка экрана
                Clear();

                // Получение входных данных

```

```

    Get_Input();

    // Выполнение программы логики и
    // искусственного интеллекта
    Do_Logic();

    // Отображение следующего кадра анимации,
    // визуализация двумерного или
    // трехмерного мира
    Render_Frame();

    // Стабилизируем частоту вывода
    Wait();

    // Единственный способ изменить
    // состояние - это взаимодействие с
    // пользователем в разделе ввода данных
    // или, возможно, прерывание игры
} break;

case GAME_RESTART: // Перезапуск игры
{
    // Здесь выполняется очистка для решения
    // всех проблем для нового запуска игры
    Fixup();
    // Возврат в меню
    game_state = GAME_MENU;
} break;

case GAME_EXIT: // Выход из игры
{
    // Если игра находится в этом состоянии,
    // пришло время освободить ресурсы и
    // выйти из игры
    Release_And_Cleanup();

    // Устанавливаем код ошибки
    error = 0;

    // Примечание: переключать состояние не
    // требуется, т.к. на следующей стадии
    // код выходит из основного цикла
    // и возвращается в ОС
} break;

default: break;
} // switch

} // while

// Возврат кода ошибки операционной системе
return(error);

} // main

```

Хотя этот код нефункционален, я думаю, что изучив его, вы поймете идею структуры реального цикла игры. Все циклы игр — не имеет значения, трехмерных или двумерных, — довольно точно повторяют приведенную структуру. На рис. 1.2 показана диаграмма переходов между состояниями игрового цикла. Как видно, переходы между состояниями довольно последовательны.

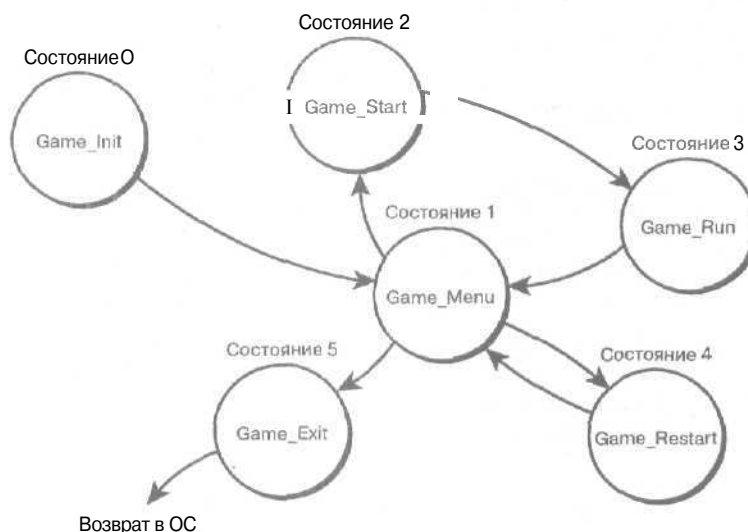


Рис. 1.2. Диаграмма переходов между логическими состояниями цикла игры

## Общие советы по программированию игр

Следующее, о чем я хочу поговорить, — это общепринятые методы и философия программирования игр, о которых вам следует подумать и которые следует принять (если, конечно, вы это сможете). Они значительно облегчают процесс программирования.

Начнем с того, что видеоигры — это компьютерные программы, требующие высокой производительности компьютера. Это означает, что для разделов кода с высоким требованием к машинному времени и памяти больше нельзя использовать высокоуровневые API. В большинстве случаев следует самому писать все, что касается внутреннего цикла игрового кода, иначе игра будет работать ужасающе медленно. Конечно, это не означает, что нельзя полагаться на такие API, как DirectX, поскольку DirectX специально написан так, чтобы быть максимально быстрым при минимальном размере кода. Но в целом следует избегать вызовов высокоуровневых функций Win32 API. Например, вы можете решить, что функция `memset()` достаточно быстрая, однако она заполняет память побайтово. Значительно лучше использовать версию функции, которая заполняет память сразу по четыре байта (одно слово). Вот пример функции на встроенном ассемблере, которую я использую для заполнения памяти по четыре байта.

```

inline void Mem_Set_QUAD(void *dest UINT data, int count)
{
    // Заполнение памяти по 32 бита. count — размер памяти
    // в четырехбайтовых словах

```

```

    _asm
    {
        mov edi, dest ; edi указывает адрес памяти
        mov ecx, count ; Количество 32-битовых слов
        mov eax, data ; 32-битовые данные
        rep stosd ; Перенос данных
    } // asm

} // Mem_Set_QUAD

    А вот версия для двухбайтового слова.

inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
    // Заполнение памяти по 16 битов. count — размер памяти
    // в двухбайтовых словах

    _asm
    {
        mov edi, dest ; edi указывает адрес памяти
        mov ecx, count ; Количество 16-битовых слов
        mov ax, data ; 16-битовые данные
        rep stosw ; Перенос данных
    } // asm

} // Mem_Set_WORD

```

Эти несколько строк кода могут в некоторых случаях ускорить игру в два или четыре раза! Поэтому вам обязательно следует знать, что находится внутри функции API, если вы собираетесь ее использовать.

#### НА ЗАМЕТКУ

Кроме того, Pentium III, 4 и последующие версии поддерживают SIMD-команды (Single Instruction Multiple Data — один поток команд и множество потоков данных), которые позволяют распараллеливать обработку простых математических операций, поэтому здесь открывается большое поле деятельности для оптимизации базовых математических операций, таких как векторная математика и умножение матриц. К этому мы вернемся позднее.

Давайте взглянем на перечень приемов, о которых следует помнить во время программирования.

#### СЕКРЕТ

Не бойтесь использовать глобальные переменные. Многие видеоигры не используют передачу параметров в критических по времени выполнения функциях, вместо этого применяя глобальные переменные. Рассмотрим, например, следующую функцию,

```

void PLOT(int x, int y, int color)
{
    // Вывод точки на экран
    video_buffer[x + y*MEMORY_PITCH] = color;
} // Plot

```

В данном случае тело функции выполняется гораздо быстрее, чем ее вызов, вследствие необходимости внесения параметров в стек и снятия их оттуда. В такой ситуации более эффективным может оказаться создание соответствующих глобальных переменных и передача информации в функцию путем присвоения им соответствующих значений,

```
int gx, gy, gcolor; // Глобальные переменные

void Plot_G(void)
{
    // Вывод точки на экран
    video_buffer[gx + gy*MEMORY_PITCH] = gcolor;
} // Plot_G
```

СЕКРЕТ

Используйте встраиваемые функции. Предыдущий фрагмент можно еще больше улучшить с помощью директивы `inline`, которая полностью устраняет код вызова функции, указывая компилятору на необходимость размещения кода функции непосредственно в месте ее вызова. Конечно, это несколько увеличивает программу, но скорость для нас гораздо важнее<sup>1</sup>.

```
inline void Plot_I(intx, inty, int color)
{
    // Вывод точки на экран
    video_buffer[x + y*MEMORY_PITCH] = color;
} // Plot_I
```

Заметим, что здесь не используются глобальные переменные, поскольку компилятор сам заботится о том, чтобы для передачи параметров не использовался стек. Тем не менее, глобальные переменные могут пригодиться, если между вызовами изменяется только один или два параметра, поскольку при этом не придется заново загружать старые значения.

СЕКРЕТ

Всегда используйте 32-битовые переменные вместо 8- или 16-битовых. Pentium и более поздние процессоры — 32-битовые, а это означает, что они хуже работают со словами данных размером 8 и 16 битов и их использование может замедлить работу в связи с эффектами кэширования и другими эффектами, связанными с адресацией памяти. Например, вы можете создать структуру, которая выглядит примерно следующим образом.

```
struct CPOINT
{
    short x, y;
    unsigned char c;
} // CPOINT
```

Хотя использование такой структуры может показаться стоящей идеей, на самом деле это вовсе не так! Эта структура имеет размер 5 байтов;  $(2 * \text{sizeof}(\text{short}) + \text{sizeof}(\text{unsigned char})) = 5$ . Это не очень хорошо в силу особенностей адресации памяти у 32-битовых процессоров. Гораздо лучше использовать следующую структуру.

```
struct CPOINT
{
    int x, y;
    int c;
} // CPOINT
```

Такая структура гораздо лучше. Все ее элементы имеют одинаковый размер — 4 байта, а следовательно, все они выровнены в памяти на границу DWORD. Несмотря на выросший размер данной структуры, работа с ней осуществляется эффективнее, чем с предыдущей.

<sup>1</sup> Теоретически возможна ситуация, когда из-за использования встроенных функций внутренний цикл может вырасти и перестать полностью помещаться в кэше процессора, что приведет к снижению производительности по сравнению с использованием обычных функций. Злоупотреблять встроенными функциями не следует, и не только по этой причине. — *Прим. ред.*

В действительности вы можете доводить размер всех своих структур до величины, кратной 32 байтам, поскольку это оптимальный размер для стандартного кэша процессоров класса Pentium. Довести размер до этого значения можно путем добавления дополнительных искусственных членов структур либо посредством соответствующих директив компилятора. Конечно же, такой рост размера структур приведет к перерасходу памяти, но он может оказаться оправданным увеличением скорости работы.

C++

struct в C++ представляет собой аналог `class`, у которого все члены по умолчанию открыты (`public`).

СЕКРЕТ

Тщательно комментируйте ваш код. Программисты, работающие над играми, пользуются дурной славой в связи с тем, что не комментируют свой код. Не повторяйте их ошибку. Ясный, хорошо комментированный код стоит лишней работы с клавиатурой.

СЕКРЕТ

Программируйте в стиле RISC. Другими словами, делайте ваш код как можно более простым. В частности, в силу особенностей архитектуры процессоры класса Pentium предпочитают простые инструкции сложным, да и компилятору работать с ними и оптимизировать их легче. Например, вместо кода

```
if ( (x += (2*buffer[index++])) > 10 )
{
    // Выполняем некоторые действия
} // if

используйте код попроще
x += 2*buffer[index];
index++;

if (x > 10)
{
    // Выполняем некоторые действия
} // if
```

На то есть две причины. Во-первых, такой стиль позволяет при отладке вставлять дополнительные точки останова между разными частями кода. Во-вторых, такой подход облегчает работу компилятора по оптимизации этого кода.

СЕКРЕТ

Вместо умножения целых чисел на степень двойки, используйте побитовый сдвиг. Поскольку данные в компьютере хранятся в двоичном виде, сдвиг влево или вправо эквивалентен, соответственно, умножению или делению, например:

```
int y_pos = 10;

// Умножаем y_pos на 64
y_pos = (y_pos << 6); // 2^6 = 64

// Делим y_pos на 8
y_pos = (y_pos >> 3); // (1/3)^3 = 1/8
```

Вы еще встретитесь с подобными советами в главе, посвященной оптимизации.

СЕКРЕТ

Используйте эффективные алгоритмы. Никакой ассемблерный код не сделает алгоритм  $O(n^2)$  более быстрым. Лучше использовать более эффективные алгоритмы, чем пытаться оптимизировать куда негодные.

**СЕКРЕТ**

Не оптимизируйте ваш код в процессе программирования. Обычно это просто пустая трата времени. Перед тем как приступить к оптимизации, завершите написание если не всей игры, то по крайней мере, основной ее части. Тем самым вы сохраните силы и сэкономите время. Только когда игра **готова, наступает** время для ее профилирования и поиска проблемных участков кода, которые следует оптимизировать.

**СЕКРЕТ**

Не используйте слишком сложные структуры данных для простых объектов. Не стоит использовать связанные списки для представления массива, количество **элементов** которого точно известно заранее, только в силу того, что такие списки — это очень круто. Программирование видеоигр на 90% состоит из работы с данными. Храните их в как можно более простом виде с тем, чтобы обеспечить как можно более быстрый и простой доступ к ним. Убедитесь, что используемые вами структуры данных **наиболее** подходят для решения ваших задач.

**СЕКРЕТ**

Разумно используйте C++. Не пытайтесь применять множественное наследование только потому, что вы знаете, **как это делается**. Используйте только те возможности, которые реально необходимы и результаты применения которых вы хорошо себе представляете.

**СЕКРЕТ**

Если вы видите, что зашли в тупик, остановитесь и без сожалений вернитесь **назад**. Лучше потерять 500 строк кода, чем получить совершенно неработоспособный проект.

**СЕКРЕТ**

Регулярно делайте резервные копии вашей работы. При работе с игровой программой достаточно легко восстановить какую-нибудь простую сортировку, но восстановить систему искусственного интеллекта — дело совсем другое.

**СЕКРЕТ**

Перед тем **как** приступить к созданию игры, четко **организу**йте свою работу. Используйте понятные и имеющие смысл имена файлов и каталогов, выберите и придерживайтесь последовательной системы **имено**вания переменных, постарайтесь разделить графические и звуковые данные по **разным** каталогам.

## Использование инструментов

Раньше для написания видеоигр обычно не требовалось ничего, кроме текстового редактора и, возможно, кустарных графических и звуковых программ. Сегодня ситуация намного **усложнилась**. Для написания трехмерной игры необходим, как минимум, компилятор C/C++, программа создания двумерных изображений, программа обработки звука, а также какой-нибудь минимальный разработчик трехмерных объектов (если только вы не собираетесь вводить все **3D-модели** как ASCII-данные). Кроме того, понадобится музыкальный секвенсор, если вы собираетесь использовать какие-либо **MIDI-данные**.

Давайте рассмотрим некоторые наиболее популярные программы и их возможности.

- **Компиляторы C/C++.** Для работы под Windows 9x/Me/XP/2000/NT нет лучшего компилятора, чем MS VC++ (рис. 1.3). Он делает все, что может вам потребоваться, и даже больше того. Генерируемые им **.EXE-файлы** обладают максимально быстрым кодом. Компилятор фирмы Borland также хорошо работает, однако имеет меньше возможностей. В любом случае вам не нужна полная версия — достаточно студенческой версии, способной генерировать Win32 **.EXE-файлы**.
- **Программы для двумерной растровой графики.** Сюда относятся программы для рисования, черчения и обработки изображений. Программы для рисования позволяют рисовать изображения попиксельно с помощью примитивов и редактировать их. Насколько мне известно, Paint Shop **Pro** фирмы JASC

(рис. 1.4) является лидером по показателю соотношения цены и производительности. Но все же несомненным фаворитом большинства компьютерных художников является Adobe Photoshop ~ он обладает гораздо большей мощностью, чем та, которая когда-либо может вам понадобиться.



Рис. 1.3. Среда Microsoft VC++

- **Программы для двумерной векторной графики.** Эти программы позволяют создавать изображения, которые состоят в основном из кривых, прямых и двумерных геометрических примитивов. Этот тип программ не столь полезен, однако вам потребуется одна из них. Adobe **Illustrator** — это то, что вам нужно,
- **Программы для завершающей обработки изображений.** Редакторы изображений — это финальный класс программ для двумерной графики. Эти программы предназначены больше для завершающей обработки, чем для создания изображений. В этой области фаворитом многих пользователей является Adobe Photoshop, однако мне больше по душе Corel Photo-Paint. Решать вам.
- **Программы для обработки звука.** 90% всех звуковых эффектов, используемых в играх, — это оцифрованный звук. Для работы со звуковыми данными этого типа вам понадобится программа обработки звука. Одна из лучших программ этого рода — Sound Forge (рис. 1.5). Несомненно, это одна из самых сложных программ обработки звука. Я все еще открываю в ней новые и новые замечательные возможности! Еще одна довольно мощная программа — Cool Edit Pro, но с ней я работал меньше.
- **Программы разработки трехмерных объектов.** Здесь все зависит от ваших финансовых возможностей. Такие программы могут стоить десятки тысяч долларов. Тем не менее, опыт показывает, что последние версии не очень дорогих редакторов трехмерных объектов могут в буквальном смысле создавать кино. Для разработки простых и среднего уровня сложности трехмерных моделей я использую, главным образом, Caligari trueSpace (рис. 1.6). Это самый лучший 3D-редактор по такой цене — он стоит всего несколько сот долларов. Кроме того, я считаю, что у него самый удобный интерфейс.

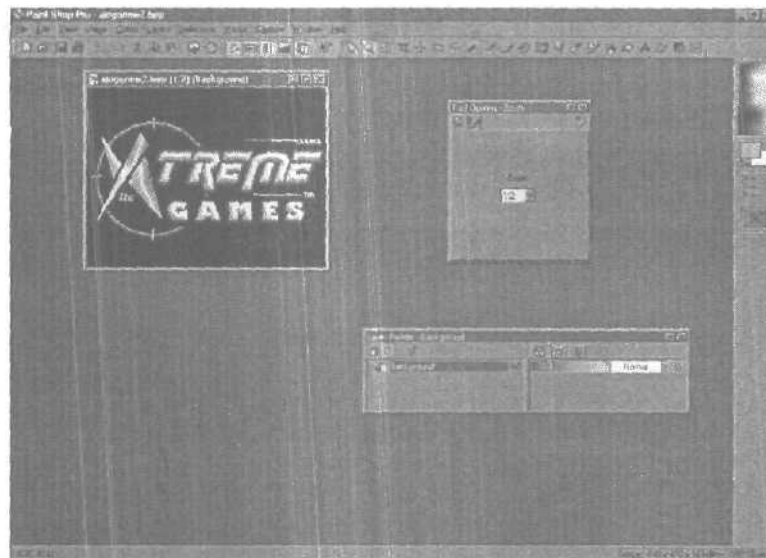


Рис. 1.4. Paint Shop Pro фирмы JASC Software



Рис. 1.5. Sound Forge в работе

Если же вам недостаточно этих возможностей и требуется полный фотореализм — используйте 3D Studio Max (рис. 1.7). Однако он стоит 2500 долл., так что тут есть о чем задуматься. В то же время, поскольку мы собираемся использовать эти программы в большинстве случаев лишь для создания трехмерных сеток, а не для визуализации, все эти “рюшечки и финтифлюшечки” нам не нужны, и наилучшим выбором будет trueSpace или, возможно, какая-то бесплатная или условно-бесплатная программа.

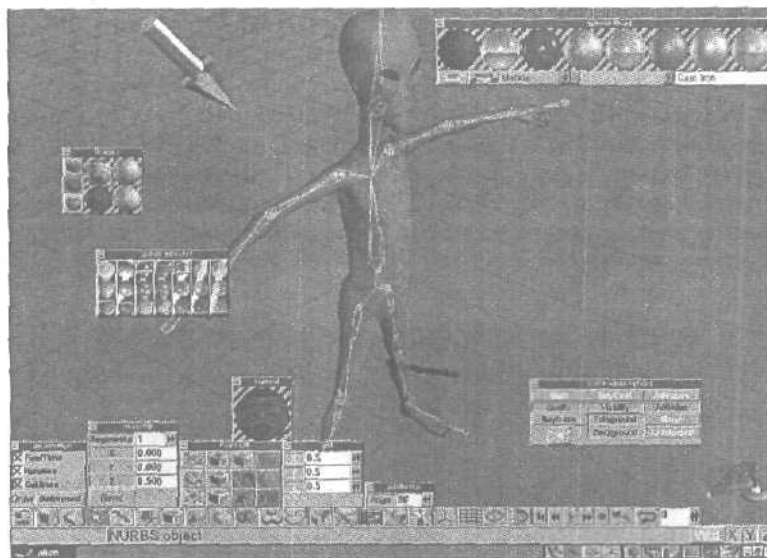


Рис. 1.6. Редактор Caligari trueSpace

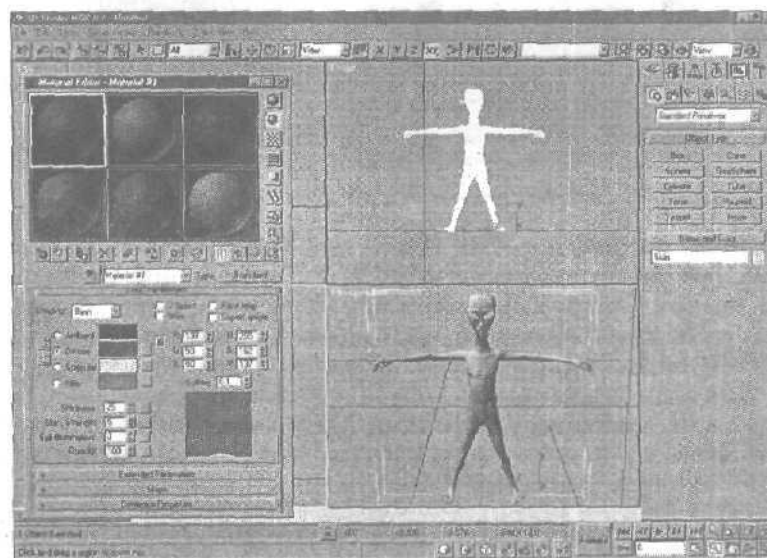


Рис. 1.7. Тяжеловес 3D Studio Max

#### НА ЗАМЕТКУ

В Internet можно найти множество бесплатных или условно-бесплатных 3D-редакторов, которые иногда имеют такие же возможности, как и коммерческий продукт. Поэтому, если денежные средства ограничены — ищите, и, возможно, вы найдете то, что нужно.

## Редакторы трехмерных уровней

В этой книге мы собираемся создать программный трехмерный игровой процессор. Однако мы не намерены создавать сложные инструменты для моделирования трехмер-

ного интерьера. Трехмерный мир можно **создать** с помощью редактора 3D-объектов, однако есть программы, гораздо лучше приспособленные для этого, такие как **WorldCraft** (рис. 1.8). Следовательно, самый правильный подход при написании трехмерной игры и игрового процессора — это использовать файловый формат и структуру данных, совместимые с наиболее доступным из редакторов, таким как **WorldCraft** (или подобным). В этом случае для **создания** миров вы сможете воспользоваться инструментами сторонних разработчиков. Конечно, файловый формат большинства таких редакторов основан преимущественно на разработках фирмы id Software и игрового процессора Quake. Тем не менее, будет ли это файловый формат Quake или какой-либо иной, в любом случае очевидно, что этот формат проверен и работает прекрасно.

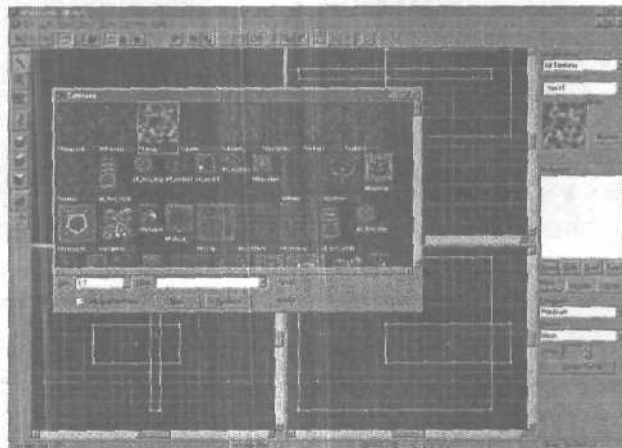


Рис. 1.8. Редактор уровней **WorldCraft**

- **Музыкальные программы и MIDI-секвенсоры.** В современных играх есть два типа музыки: цифровая (как на компакт-дисках) и **MIDI-музыка** (**Musical Instruments Digital Interface** — цифровой интерфейс музыкальных инструментов), представляющая собой синтетический объект, основанный на нотных данных. Для управления **MIDI-данными** необходим секвенсор. Один из лучших — **CakeWalk** (рис. 1.9). Одним из достоинств **CakeWalk** является приемлемая цена. Поэтому, если вы намерены работать с **MIDI-музыкой**, я весьма рекомендую познакомиться с этой программой. Мы поговорим о **MIDI-данных**, когда речь пойдет о **DirectMusic**.

#### НА ЗАМЕТКУ

А сейчас кое-то на десерт... Некоторые производители программ, упоминавшихся выше, разрешили мне записать на компакт-диск свои условно-бесплатные или ознакомительные версии. Ознакомьтесь с ними!

## Использование компилятора

Один из наиболее напряженных моментов в изучении программирования трехмерных игр — это использование компилятора. В большинстве случаев вы настолько переполнены желанием начать работу, что входите в интегрированную среду разработки и запускаете компилятор. И получаете миллионы ошибок компилирования и связывания! Чтобы решить проблему, рассмотрим несколько основных **принципов** компилирования.

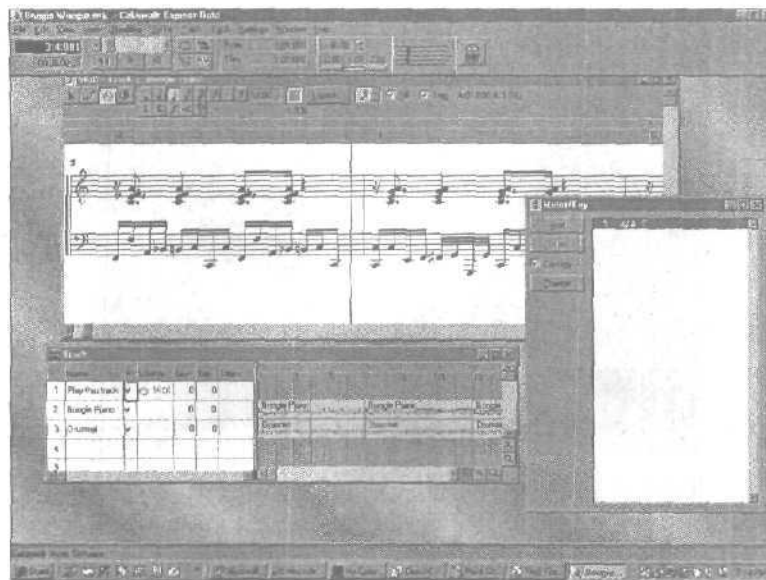


Рис. 1.9. Программа CakeWalk

**НА ЗАМЕТКУ**

Излагаемые здесь сведения относятся к Visual C++, однако на принципиальном уровне они касаются всех компиляторов.

1. Прочтите руководство пользователя компилятора полностью — пожалуйста, я умоляю вас!
2. Вы должны установить в системе DirectX 9.0 SDK. Чтобы сделать это, зайдите в каталог DirectX SDK на компакт-диске, прочтите `README.TXT` и выполните все, что там написано, т.е. щелкните на файле запуска установки DirectX SDK. И помните, в этой книге мы используем DirectX 9.0 (однако пользуемся интерфейсами версий 7 и 8), так что вам нужно использовать DirectX SDK с компакт-диска. Хотя — если вы действительно этого хотите — можете компилировать все с помощью DirectX 8.0.
3. Мы собираемся создавать Win32 .EXE-модули приложений, а не DLL, не компоненты ActiveX, не консольные приложения и не что-либо еще (если только я не скажу вам об этом отдельно). Поэтому первое, что нужно сделать, это создать **новый** проект и установить в качестве выходного файла Win32 .EXE. На рис. 1.10 показано, как сделать этот шаг в компиляторе Visual C++ 6.0.

**НА ЗАМЕТКУ**

Есть два типа .EXE-модулей, которые можно компилировать: `Release` и `Debug`. Окончательный (`Release`) обладает более быстрым и оптимизированным кодом, тогда как отладочный (`Debug`) медленнее и содержит контрольные точки отладчика. Советую во время разработки программы использовать `Debug`-версию, а затем, когда программа заработает, переключить компилятор в режим `Release`.

4. Для добавления исходных файлов в проект используйте команду `Add Files` из главного меню или из самого узла проекта. Этот шаг для Visual C++ 6.0 показан на рис. 1.11.

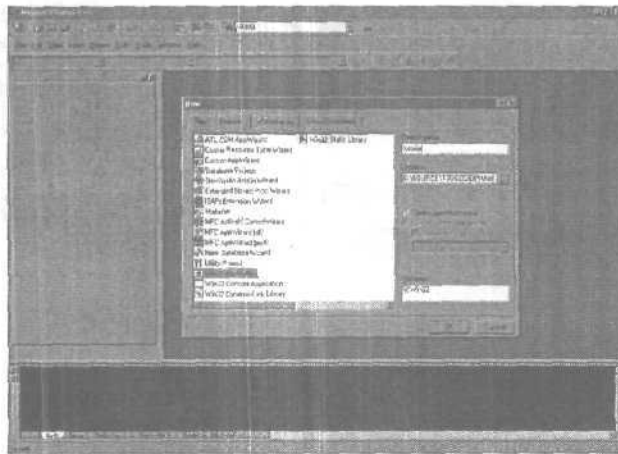


Рис. 1.10. Создание Win32 .EXE-модуля с помощью Visual C++ 6.0



Рис. 1.11. Добавление файлов в проект в среде Visual C++ 6.0

5. Когда вы компилируете что-то, для чего нужен DirectX, необходимо указать в пути поиска файлов компилятором путь к месту нахождения заголовочных файлов DirectX, а также .LIB-файлов DirectX. Это можно сделать с помощью пункта меню Options, Directories в главном меню интегрированной среды разработки. Затем сделайте соответствующие добавления в переменные Include Files и Library Files. Этот шаг показан на рис. 1.12 (конечно, следует убедиться, что в них указан путь к конкретному месту инсталляции DirectX, а также новейших версий библиотек и заголовочных файлов).

**НА ЗАМЕТКУ**

Проверьте, чтобы узлы поиска DirectX были первыми в списке(ax). Вы же не хотите, чтобы компилятор нашел старые версии файлов DirectX, которые могли быть установлены вместе с компилятором?

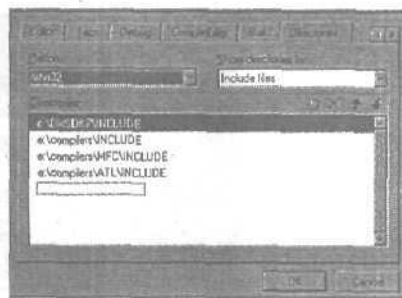


Рис. 1.12. Настройка пути поиска файлов в Visual C++ 6.0

6. Кроме того, вам обязательно нужно включить в проект библиотеки импорта COM-интерфейса DirectX, перечисленные ниже и показанные на рис. 1.13.

- `DDRAW.LIB`
- `DSOUND.LIB`
- `DINPUT.LIB`
- `DINPUT8.LIB`
- А также другие, которые я упоминаю в конкретных примерах.

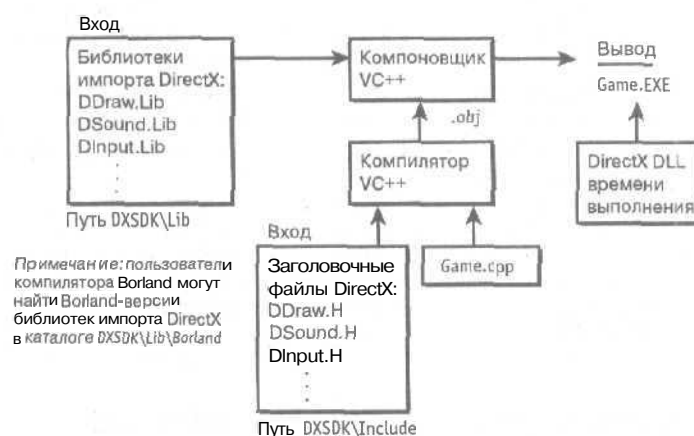


Рис. 1.13. Ресурсы, необходимые для создания приложения Win32 DirectX

Эти `.LIB`-файлы DirectX расположены в каталоге `LIB\`, находящемся там, куда был установлен DirectX SDK. Вам необходимо добавить эти `.LIB`-файлы в свой проект. Однако нельзя просто добавлять каталог `LIB\` в путь поиска — тем самым компилятору/компоновщику указывается место поиска, но не говорится о том, что нужно использовать именно DirectX `.LIB`-файлы. Я получил тысячи (действительно тысячи!) электронных писем от людей, которые не сделали этого и получили проблемы. Повторю еще раз: вам необходимо вручную включить `.LIB`-файлы DirectX в список связывания компилятора вместе с другими библиотеками или в проект вместе с вашими `.CPP`-файлами. Это можно сделать в подменю Project, Settings диалогового окна Link, General в списке Object/Library-Modules (рис. 1.14).

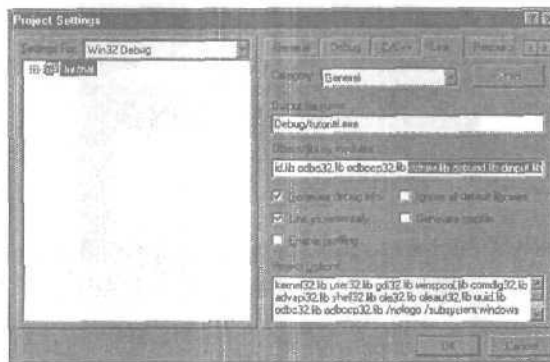


Рис. 1.14. Добавление .LIB-файлов DirectX в список связывания Visual C++ 6.0

#### НА ЗАМЕТКУ

Если вы используете Visual C++, можете добавить в проект библиотеку расширений Multimedia Windows (WINMM.LIB). Этот файл находится в каталоге LIB\ в месте установки компилятора Visual C++. Если его там нет, воспользуйтесь командой Find меню Start. Найдя файл, добавьте его в список связывания.

7. Теперь вы готовы к компиляции программ.

#### НА ЗАМЕТКУ

Для пользователей Borland в DirectX SDK есть каталог BORLAND\ . Убедитесь, что добавлены именно эти .LIB-файлы, а не файлы DirectX, установленные вместе с Visual C++.

Если у вас есть еще вопросы по этой теме, не волнуйтесь — на протяжении книги при компилировании программ я буду **возвращаться** к этим операциям много раз. Однако если я получу письмо с вопросами по компилированию и меня будут спрашивать о том, о чем только что говорилось, я за себя не ручаюсь!

#### НА ЗАМЕТКУ

Я уверен, что вы слышали о новой системе Visual .NET. Технически это компилятор и технология программирования Microsoft под новой маркой. Система нормально работает только на Windows XP/2000 и избыточна для наших целей, поэтому мы остановимся на Visual C++ 6.0. Тем не менее, все работает и при использовании .NET: Win32 есть Win32.

## Пример трехмерной игры: Raiders 3D

Прежде чем у нас начнут плавиться мозги от разговоров о математике, программировании трехмерных игр и графике, я бы хотел сделать паузу и показать уже готовую трехмерную космическую игру — простую, но все же игру. При этом вы **увидите**, что такое настоящий игровой цикл, некоторые вызовы графических функций и компиляция. Звучит неплохо, не правда ли?

Проблема состоит в том, что это только первая глава, поэтому я не могу использовать материал из последующих глав — это было бы нечестно, согласны? И я еще не показал вам игровой процессор из предыдущей книги. Поэтому я решил приучить вас **пользоваться** для программирования игр API как "черным ящиком". И эта простая игра позволит вам привыкнуть к идее использования такого API, как DirectX и игрового процессора из первой книги.

Хотя в 70-х, 80-х годах и в начале 90-х годов прошлого столетия можно было все делать самому, в 21 веке в компьютере функционирует слишком много подсистем оборудования и программного обеспечения, что делает практически невозможным написание всего кода одним человеком. Увы, мы всего лишь разработчики игр. и наш удел — использовать чужие API, такие как Win32, DirectX и т.п.

Основываясь на принципе "черного ящика", я поставлю вопрос так: каков абсолютный минимум средств для создания 16-разрядной трехмерной каркасной космической игры?

Все, что нам необходимо от API — это следующая функциональность:

- переключение в любой графический режим с помощью DirectX;
- рисование цветных линий и пикселей на экране;
- получение ввода клавиатуры;
- проигрывание звуковой записи из WAV-файла, находящегося на диске;
- проигрывание MIDI-музыки из MID-файла, находящегося на диске;
- синхронизация игрового цикла с помощью функций работы со временем;
- вывод на экран строк цветного текста;
- копирование на экран двойного буфера (внеэкранный страницы визуализации).

Вся эта функциональность, конечно, содержится в библиотеке игровых модулей T3DLIB\*, созданной в предыдущей книге и состоящей из шести файлов: T3DLIB1.CPP;H, T3DLIB2.CPP;H, и T3DLIB3.CPP;H. Модуль 1 предназначен для базовой работы с DirectX, модуль 2 — с DirectInput и DirectSound, а модуль 3 — в основном с DirectMusic.

Основываясь на использовании минимального набора функций из игровой библиотеки T3DLIB\*, я написал игру под названием Raiders 3D, которая демонстрирует ряд концепций, уже обсуждавшихся в данной главе. Кроме того, поскольку это каркасно-трехмерная игра, массу кода вообще не имеет смысла писать — и это хорошо! Хотя я буду кратко объяснять используемые математику и алгоритмы, не тратьте много времени, пытаясь понять, что к чему: эта информация дается только для того, чтобы позволить шире взглянуть на вещи.

Raiders 3D иллюстрирует все основные компоненты реальной трехмерной игры, включая игровой цикл, вычисления, искусственный интеллект, определение столкновений, звук и музыку. На рис. 1.15 показана копия экрана работающей игры. Конечно, это не Star Wars, однако совсем неплохо для нескольких часов работы!

Прежде чем показать исходный код игры, я хочу, чтобы вы получили представление о том, из каких компонентов состоит проект (рис. 1.16). Как видно из рисунка, игра состоит из следующих файлов, являющихся частью проекта:

- RAIDERS3D.CPP — основной блок логики игры, который использует функциональность T3DLIB и создает минимальное Win32-приложение;
- T3DLIB1.CPP — исходные файлы библиотеки T3DLIB;
- T3DLIB2.CPP;
- T3DLIB3.CPP;
- T3DLIB1.H — заголовочные файлы библиотеки;
- T3DLIB2.H;
- T3DLIB3.H;
- DDRAW.LIB — DirectDraw — компонент двумерной графики в составе DirectX. Для создания приложения требуется библиотека импорта DDRAW.LIB. Она не содержит

код DirectX; это промежуточная библиотека, обеспечивающая обращение к динамической библиотеке DDRAW.DLL, которая выполняет реальную работу. Этот файл можно найти в DirectX SDK в каталоге LIB\;

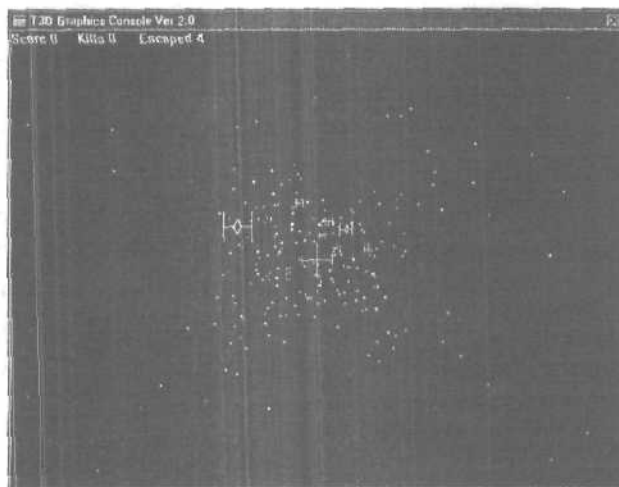


Рис. 1.15. Копия экрана Raiders 3D

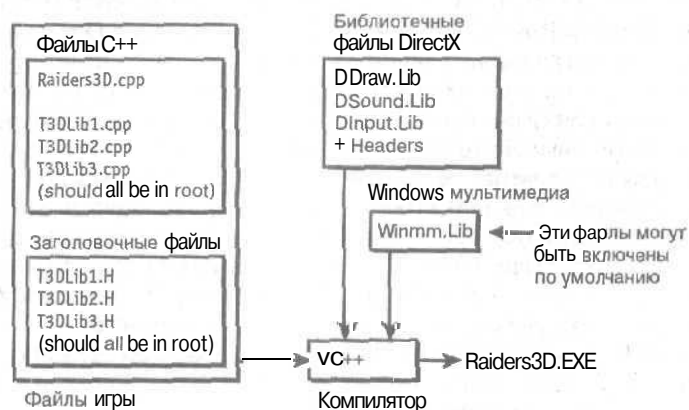


Рис. 1.16. Структура кода Raiders 3D

- **DINPUT.LIB/DINPUT8.LIB** — **DirectInput** — компонент пользовательского ввода в составе DirectX. Для создания приложения требуются библиотеки импорта **DINPUT.LIB** и **DINPUT8.LIB**;
- **DSOUND.LIB** — **DirectSound** — компонент цифрового звука в составе DirectX. Для создания приложения требуется библиотека импорта **DSOUND.LIB**.

**НА ЗАМЕТКУ**

Отметим, что не существует файла **DMUSIC.LIB**, хотя **DirectMusic** используется в **T3DLIB**. Это так, поскольку **DirectMusic** — это чистый COM-объект. Иными словами, нет библиотеки импорта, содержащей интерфейсные функции для обращения к **DirectMusic** — вам нужно все делать самому. К счастью, я уже сделал этот для вас!

Следующие файлы не нужны компилятору или компоновщику, но они являются выполняемыми DLL-файлами DirectX, которые загружаются при запуске игрового приложения:

- **DINPUT.DLL/DINPUT8.DLL** — это динамически компокуемые библиотеки **DirectDraw**, которые содержат **COM-реализацию** функций интерфейса **DirectInput**, вызываемых через библиотеку импорта **DINPUT.LIB**. Здесь вам не нужно ни о чем беспокоиться, проследите лишь, чтобы были установлены исполняемые файлы DirectX;
- **DDRAW.DLL** — это динамически компокуемая библиотека **DirectDraw**, которая содержит **COM-реализацию** интерфейсных функций **DirectDraw**, вызываемых через библиотеку импорта **DDRAW.LIB**;
- **DSOUND.DLL** — это динамически компокуемая библиотека **DirectSound**, которая содержит **COM-реализацию** интерфейсных функций **DirectSound**, вызываемых через библиотеку импорта **DSOUND.LIB**;
- **DMUSIC.DLL** — это динамически компокуемая библиотека **DirectMusic**, которая содержит **COM-реализацию** интерфейсных функций **DirectMusic**, вызываемых непосредственно через **COM-обращения**.

Основываясь на нескольких вызовах библиотеки, я создал игру **RAIDERS3D.CPP**, показанную в нижеприведенном листинге. Игра запускается в оконном режиме с 16-разрядной графикой, поэтому проследите, чтобы экран находился в 16-битовом цветовом режиме.

Хорошо изучите представленный код: главный цикл игры, трехмерную математику, а также вызовы других функций.

```
// Raiders3D - RAIDERS3D.CPP - наша первая трехмерная игра
// ПРОЧИТЕ ЭТО!
// Перед компиляцией убедитесь, что включили в список
// связывания проекта файлы DDRAW.LIB, DSOUND.LIB,
// DINPUT.LIB, WINMM.LIB, а также исходные модули
// T3DLIB1.CPP, T3DLIB2.CPP и T3DLIB3.CPP в проект!!!
// Включите заголовочные файлы T3DLIB1.H, T3DLIB2.H и
// T3DLIB3.H в рабочий каталог, с тем чтобы компилятор
// мог найти их.

// Для запуска игры убедитесь, что экран установлен в
// 16-битовый цветовой режим с разрешением 640x480 или выше

// INCLUDES ////////////////////////////////////////

#define INITGUID // Гарантируем доступность COM-интерфейсов
                // Вместо этого можно подключить .LIB-файл
                // DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h> // Подключаем функциональность Windows
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // Подключаем функциональность C/C++
#include <conio.h>
#include <stdlib.h>
```

```

#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h> // Подключаем DirectX
#include <dsound.h>
#include <dmsctrl.h>
#include <dmusic.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <diinput.h>
#include "T3DLIB1.h" // Подключаем библиотеку T3D
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// DEFINES ////////////////////////////////////////

// Определяем интерфейс windows
typedef WINDOW_CLASS_NAME "WIN3DCLASS" // Имя класса
#define WINDOW_TITLE "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH 640 // Размер окна
#define WINDOW_HEIGHT 480

#define WINDOW_BPP 16 // Битовая глубина цвета окна
// (8,16,24 и т.д.)
// Примечание: если работа идет в окне и не используется
// полноэкранный режим работы, то битовая глубина цвета
// должна быть такой же, как в системе. То же самое в
// случае создания и подключения 8-разрядной палитры

#define WINDOWED_APP 1 // 0 - не оконный режим, 1 - оконный

//////////////////////////////////////

#define NUM_STARS 250 // Число звезд в модели
#define NUM_TIES 10 // Число боевых кораблей в модели

// Константы 3D-игрового процессора
#define NEAR_Z 10 // Ближняя плоскость отсечения
#define FAR_Z 2000 // Дальняя плоскость отсечения
#define VIEW_DISTANCE 320 // Расстояние видимости для данной
// точки обзора. Оно дает размер
// поля зрения при угле зрения 90
// в случае проекции на окно
// шириной 640 пикселей

// Константы игрока
#define CROSS_VEL 8 // Скорость, с которой движется
// перекрестие прицела

```

```

#define PLAYER_Z_VEL 8 // виртуальная z-скорость игрока
                        // для имитации перемещения

// Константы модели боевого корабля
#define NUM_TIE_VERTS 10
#define NUM_TIE_EDGES 8

// Данные о взрыве
#define NUM_EXPLOSIONS (NUM_TIES) // Общее число взрывов

// Состояния игры
#define GAME_RUNNING 1
#define GAME_OVER 0

// ТИПЫ //////////////////////////////////////

// Трехмерная точка
typedef struct POINT3D_TYP
{
    USHORT color; // 16-битовый цвет точки
    float x,y,z; // Координаты точки
} POINT3D, *POINT3D_PTR;

// 3D-линия, два индекса в списке вершин
typedef struct LINE3D_TYP
{
    USHORT color; // 16-битовый цвет линии
    int v1,v2; // Индексы конечных точек в списке вершин
} LINE3D, *LINE3D_PTR;

// Боевой корабль
typedef struct TIE_TYP
{
    int state; // Состояние боевого корабля:
                // 0=мертв, 1=жив
    float x, y, z; // Координаты корабля
    float xv,yv,zv; // Скорость корабля
} TIE, *TIE_PTR;

// Базовый 3D-вектор, используемый для скорости
typedef struct VEC3D_TYP
{
    float x,y,z; // Координаты вектора
} VEC3D, *VEC3D_PTR;

// Каркасный взрыв
typedef struct EXPL_TYP
{
    int state; // Состояние взрыва
    int counter; // Счетчик взрывов
    USHORT color; // Цвет взрыва

```

```

// Взрыв - это совокупность ребер/линий,
// основанная на модели взрывающегося корабля
POINT3D p1[NUM_TIE_EDGES]; // Начальная точка ребра n
POINT3D p2[NUM_TIE_EDGES]; // Конечная точка ребра n
VEC3D vel[NUM_TIE_EDGES]; // Скорость осколков

} EXPL, *EXPL_PTR;

// ПРОТОТИПЫ //////////////////////////////////////

// Консоль игры
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// Функции игры
void Init_Tie(int index);

// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ //////////////////////////////////////

HWND main_window_handle = NULL; // Дескриптор окна
HINSTANCE main_instance = NULL; // Сохраняем экземпляр
char buffer[256]; // Используется для
// вывода текста

// Корабль - это совокупность вершин, соединенных
// линиями, которые образуют форму

POINT3D tie_vlist[NUM_TIE_VERTS]; // Список вершин модели
// боевого корабля
LINE3D tie_shape[NUM_TIE_EDGES]; // Список ребер корабля
TIE ties[NUM_TIES]; // Боевые корабли

POINT3D stars[NUM_STARS]; // Звездное поле

// Некоторые цвета мы не можем создать, пока не знаем формат
// цвета — 5.5.5 или 5.6.5, поэтому мы подождем немного
// и сделаем это в функции Game_Init()
USHORT rgb_green,
       rgb_white,
       rgb_red,
       rgb_blue;

// Переменные игрока
float cross_x = 0, // Перекрестие прицела
      cross_y = 0;

int cross_x_screen = WINDOW_WIDTH/2, // Перекрестие прицела
    cross_y_screen = WINDOW_HEIGHT/2,
    target_x_screen = WINDOW_WIDTH/2,
    target_y_screen = WINDOW_HEIGHT/2;

int player_z_vel = 4; // Виртуальная скорость

```

```

        // наблюдателя/корабля
int cannon_state = 0; // Состояние лазерной пушки
int cannon_count = 0; // Счетчик выстрелов из пушки

EXPL explosions[NUM_EXPLOSIONS]; // Взрывы

int misses = 0; // Число кораблей, которые
        // не удалось поразить
int hits = 0; // Число попаданий
int score = 0; // Счет

// Музыкальные и звуковые данные
int main_track_id = -1, // Идентификатор музыкальной дорожки
    laser_id = -1, // Звук лазерной вспышки
    explosion_id = -1, // Звук взрыва
    flyby_id = -1; // Звук пролетающего корабля

int game_state = GAME_RUNNING; // Состояние игры

// ФУНКЦИИ //////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    // Обработчик сообщений системы
    PAINTSTRUCT ps; // Используется в WM_PAINT
    HDC hdc; // Контекст устройства

    // Какое сообщение получено?
    switch(msg)
    {
        case WM_CREATE:
        {
            // Инициализация данных
            return(0);
        } break;

        case WM_PAINT:
        {
            // Начало рисования
            hdc = BeginPaint(hwnd,&ps);

            // Окончание рисования
            EndPaint(hwnd,&ps);
            return(0);
        } break;

        case WM_DESTROY:
        {
            // Закрываем приложение

```

```

        PostQuitMessage(0);
        return (0);
    } break;

    default: break;

} // switch

// Обработка необработанных сообщений
return (DefWindowProc(hwnd, msg, wParam, lParam));

} // WinProc

// WINMAIN //////////////////////////////////////////////////////////////////////

int WINAPI WinMain( HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
                   LPSTR lpcmdline,
                   int ncmdshow)
{
    WNDCLASS winclass; // Создаваемый класс
    HWND  hwnd; // Обобщенный дескриптор окна
    MSG  msg; // Обобщенное сообщение
    HDC  hdc; // Дескриптор контекста устройства
    PAINTSTRUCT ps; // Структура вывода

    // Заполняем структуру класса
    winclass.style = CS_DBLCLKS | CS_OWNDC |
                    CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = hinstance;
    winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IOC_ARROW);
    winclass.hbrBackground =
        (HBRUSH)GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;

    // Регистрируем класс окна
    if (!RegisterClass(&winclass))
        return (0);

    // Создаем окно. Обратите внимание на проверку
    // для выбора надлежащего флага окна
    if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // Класс
                             WINDOW_TITLE, // Заголовок
                             (WINDOWED_APP ?
                              WS_OVERLAPPED | WS_SYSMENU) :
                              (WS_POPUP | WS_VISIBLE)),
        0,0, // x,y

```

```

        WINDOW_WIDTH, // Ширина
        WINDOW_HEIGHT, // Высота
        NULL // Дескриптор родителя
        NULL, // Дескриптор меню
        hinstance, // экземпляр
        NULL))) // Параметры

return(0);

// Сохраняем дескриптор и экземпляр
// окна в глобальной переменной
main_window_handle = hwnd;
main_instance = hinstance;

// Изменим окно, чтобы клиентская область имела
// размер width x height
if (WINDOWED_APP)
{
    // Изменим размер окна так, чтобы клиентская область
    // имела именно тот размер, который указан в запросе,
    // поскольку, если приложение работает в окне, там
    // могут быть рамки и панели органов управления.
    // Если приложение работает не в окне, это не имеет
    // значения
    RECT window_rect = {0,0,WINDOW_WIDTH,WINDOW_HEIGHT};

    // Вызов для настройки window_rect
    AdjustWindowRectEx(&window_rect,
        GetWindowStyle(main_window_handle),
        GetMenu(main_window_handle) != NULL
        GetWindowExStyle(main_window_handle));

    // Сохраним глобальные переменные смещения клиента,
    // необходимые для DDraw_Flip()
    window_client_x0 = -window_rect.left;
    window_client_y0 = -window_rect.top;

    // Изменим размер окна с помощью вызова MoveWindow()
    MoveWindow(main_window_handle,
        CW_USEDEFAULT, // Координата x
        CW_USEDEFAULT, // Координата y
        // Ширина и высота
        window_rect.right - window_rect.left,
        window_rect.bottom - window_rect.top,
        FALSE);

    // Выводим окно
    ShowWindow(main_window_handle, SW_SHOW);
} // if windowed

// Выполняем инициализацию параметров игровой консоли
Game_Init();

// Отключаем CTRL-ALT-DEL, ALT-TAB. Закомментируйте

```

```

// эту строку, если она вызывает зависание системы
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);

// Входим в главный цикл событий
while(1)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // Проверяем, не следует ли выйти
        if (msg.message == WM_QUIT)
            break;

        // Транслируем горячие клавиши
        TranslateMessage(&msg);

        // Посылаем сообщение обработчику окна
        DispatchMessage(&msg);
    } // if

    // Основная работа игры выполняется здесь
    Game_Main();

} // while

// Выходим из игры и освобождаем все ресурсы
Game_Shutdown();

// Включаем CTRL-ALT-DEL, ALT-TAB, закомментируйте
// эту строку, если она приводит к зависанию системы
SystemParametersInfo(SPI_SCREENSAVERRUNNING,
    FALSE, NULL, 0);

// Возвращаемся в Windows
return(msg.wParam);

} // WinMain

// ФУНКЦИИ КОНСОЛИ ИГРЫ T3D II //////////////////////////////////////

int Game_Init(void *parms)
{
    // Инициализация параметров игры

    int index;

    Open_Error_File("error.tort");
    // Запускаем DirectDraw (заменяем параметры по желанию)
    DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP,
        WINDOWED_APP);

    // Инициализируем DirectInput
    DInput_Init();

```

```

// Запрашиваем клавиатуру
DInput_Init_Keyboard();

// Инициализируем DirectSound
DSound_Init();

// Загружаем звуковые файлы
explosion_id = DSound_Load_WAV("exp1.wav");
laser_id = DSound_Load_WAV("shocker.wav");

// Инициализируем DirectMusic
DMusic_Init();

// Загружаем и запускаем музыкальный трек
main_track_id = DMusic_Load_MIDI("midifile2.mid");
DMusic_Play(main_track_id);

// Добавляем вызовы для ввода данных из других
// устройств DirectInput

// Прячем мышь
ShowCursor(FALSE);

// Инициализируем генератор случайных чисел
srand(Start_Clock());

// Создаем системные цвета
rgb_green = RGB16Bit(0,31,0);
rgb_white = RGB16Bit(31,31,31);
rgb_blue = RGB16Bit(0,0,31);
rgb_red = RGB16Bit(31,0,0);

// Создаем звездное поле
for (index=0; index < NUM_STARS; index++)
{
    // В беспорядке располагаем звезды в цилиндре,
    // вытянутом от точки наблюдателя (0,0,-d) в
    // направлении отсекающей плоскости (0,0,far_z)
    stars[index].x = -WINDOW_WIDTH/2 +
        rand()%WINDOW_WIDTH;
    stars[index].y = -WINDOW_HEIGHT/2 +
        rand()%WINDOW_HEIGHT;
    stars[index].z = NEAR_Z +
        rand()%(FAR_Z - NEAR_Z);

    // устанавливаем цвета звезд
    stars[index].color = rgb_white;
} // for index

// Создаем модель боевого корабля

// Список вершин боевого корабля
POINT3D temp_tie_vlist[NUM_TIE_VERTS] -

```

```

    // Цвет, x,y,z
    {
    {rgb_white,-40, 40,0}, // p0
    {rgb_white,-40, 0,0}, // p1
    {rgb_white,-40,-40,0}, // p2
    {rgb_white,-10, 0,0}, // p3
    {rgb_white, 0, 20,0}, // p4
    {rgb_white, 10, 0,0}, // p5
    {rgb_white, 0,-20,0}, // p6
    {rgb_white, 40, 40,0}, // p7
    {rgb_white,40, 0,0}, // p8
    {rgb_white,40,-40,0}}; // p9

    // Копируем модель в глобальные массивы
    for (index=0; index<NUM_TIE_VERTS; index++)
        tie_vlist[index] = temp_tie_vlist[index];

    // Список ребер боевого корабля
    LINE3D temp_tie_shape[NUM_TIE_EDGES] -
        // цвет, вершина 1, вершина 2
    {
    {rgb_green,0,2 }, // l0
    {rgb_green,1,3 }, // l1
    {rgb_green,3,4 }, // l2
    {rgb_green,4,5 }, // l3
    {rgb_green,5,6 }, // l4
    {rgb_green,6,3 }, // l5
    {rgb_green,5,8 }, // l6
    {rgb_green,7,9 } }; // l7

    // Копируем модель в глобальные массивы
    for (index=0; index<NUM_TIE_EDGES; index++)
        tie_shape[index] = temp_tie_shape[index];

    // Инициализируем положение каждого боевого
    // корабля и его скорость
    for (index=0; index<NUM_TIES; index++)
    {
        // Инициализируем корабль
        Init_Tie(index);
    } // for index

    // Возвращаем код успешного завершения
    return(1);

} // Game_Init

////////////////////////////////////

int Game_Shutdown(void *parms)
{
    // Эта функция выполняется при выходе из игры
    // и освобождении всех выделенных ресурсов

```

```

// Завершаем все процессы

// Освобождаем все ресурсы, выделенные для игры

// Закрываем DirectSound
DSound_Stop_All_Sounds();
DSound_Shutdown();

// DirectMusic
DMusic_Delete_All_MIDI();
DMusic_Shutdown();

// DirectInput
DInput_Shutdown();

// DirectDraw завершаем последним
DDraw_Shutdown();

// Возвращаем код успешного завершения
return(1);
} // Game_Shutdown

////////////////////////////////////

void Start_Explosion(int tie)
{
    // Взрыв, основанный на модели пораженного корабля

    // Можно ли произвести взрыв
    for (int index=0; index < NUM_EXPLOSIONS; index++)
    {
        if (explosions[index].state==0)
        {
            // Начинаем взрыв с учетом индекса корабля

            explosions[index].state = 1; // Состояние взрыва
            explosions[index].counter = 0; // Сброс счетчика

            // Цвет взрыва
            explosions[index].color= rgb_green;

            // Делаем копию списка ребер
            for (int edge=0; edge < NUM_TIE_EDGES; edge++)
            {
                // Начальная точка ребра
                explosions[index].p1[edge].x = ties[tie].x +
                    tie_vlist[tie_shape[edge].v1].x;
                explosions[index].p1[edge].y = ties[tie].y +
                    tie_vlist[tie_shape[edge].v1].y;
                explosions[index].p1[edge].z = ties[tie].z +
                    tie_vlist[tie_shape[edge].v1].z;
            }
        }
    }
}

```

```

        // Конечная точка ребра
        explosions[index].p2[edge].x = ties[tie].x +
            tie_vlist[tie_shape[edge].v2].x;
        explosions[index].p2[edge].y = ties[tie].y +
            tie_vlist[tie_shape[edge].v2].y;
        explosions[index].p2[edge].z = ties[tie].z +
            tie_vlist[tie_shape[edge].v2].z;

        // вычисляем траекторию вектора ребра
        explosions[index].vel[edge].x = ties[tie].xv -
            8*rand()%16;
        explosions[index].vel[edge].y = ties[tie].yv -
            8*rand()%16;
        explosions[index].vel[edge].z = -3*rand()%4;

    } // for edge

    return;
} // if

} // for index

} // Start_Explosion

////////////////////////////////////

void Process_Explosions(void)
{
    // Обрабатываются все взрывы

    // Цикл по всем взрывам и выполняем их визуализацию
    for (int index=0; index<NUM_EXPLOSIONS; index++)
    {
        // Проверяем, активен ли взрыв
        if (explosions[index].state==0)
            continue;

        for (int edge=0; edge<NUM_TIE_EDGES; edge++)
        {
            // Должен быть взрыв, обновляем ребра
            explosions[index].p1[edge].x+=
                explosions[index].vel[edge].x;
            explosions[index].p1[edge].y+=
                explosions[index].vel[edge].y;
            explosions[index].p1[edge].z+=
                explosions[index].vel[edge].z;

            explosions[index].p2[edge].x+=
                explosions[index].vel[edge].x;
            explosions[index].p2[edge].y+=
                explosions[index].vel[edge].y;
            explosions[index].p2[edge].z+=
                explosions[index].vel[edge].z;
        }
    }
}

```

```

    } // for edge

    // Проверяем окончание взрыва
    if (++explosions[index].counter > 100)
        explosions[index].state =
            explosions[index].counter = 0;

    } // for index

} // Process_Explosions

////////////////////////////////////

void Draw_Explosions(void)
{
    // Этот блок кода рисует все взрывы

    // Цикл по всем взрывам и визуализация
    for (int index=0; index<NUM_EXPLOSIONS; index++)
    {
        // Проверяем, является ли взрыв активным
        if (explosions[index].state==0)
            continue;

        // Выполняем визуализацию взрыва
        // Каждый взрыв составлен из ряда ребер
        for (int edge=0; edge < NUM_TIE_EDGES; edge++)
        {
            POINT3Dp1_per, p2_per;

            // Находится ли ребро вблизи плоскости отсечения
            if (explosions[index].p1[edge].z < NEAR_Z &&
                explosions[index].p2[edge].z < NEAR_Z)
                continue;

            // Шаг 1: аксонометрическое преобразование
            // каждой конечной точки
            p1_per.x = VIEW_DISTANCE*
                explosions[index].p1[edge].x/
                explosions[index].p1[edge].z;
            p1_per.y = VIEW_DISTANCE*
                explosions[index].p1[edge].y/
                explosions[index].p1[edge].z;
            p2_per.x = VIEW_DISTANCE*
                explosions[index].p2[edge].x/
                explosions[index].p2[edge].z;
            p2_per.y = VIEW_DISTANCE*
                explosions[index].p2[edge].y/
                explosions[index].p2[edge].z;

            // Шаг 2: вычисляем экранные координаты
            int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
            int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;

```

```

int index;

// Перемещаем каждый боевой корабль
// в поле обзора наблюдателя
for (index=0; index<NUM_TIES; index++)
{
    // Может, он мертв?
    if (ties[index].state==0)
        continue;

    // Перемещаем
    ties[index].z+=ties[index].zv;
    ties[index].x+=ties[index].xv;
    ties[index].y+=ties[index].yv;

    // Проверяем ближайшую плоскость отсечения
    if (ties[index].z <= NEAR_Z)
    {
        // Переустанавливаем параметры данного корабля
        Init_Tie(index);
        misses++;
    }
}

} // for index

} // Process_Ties

////////////////////////////////////

void Draw_Ties(void)
{
    // Рисуем боевые корабли в виде трехмерных каркасов

    int index;

    // Используется для вычисления ограничивающего
    // прямоугольника корабля для обнаружении попаданий
    int bmin_x, bmin_y, bmax_x, bmax_y;

    // Рисуем каждый боевой корабль
    for (index=0; index < NUM_TIES; index++)
    {
        // Рисуем следующий боевой корабль

        // Этот корабль мертв?
        if (ties[index].state==0)
            continue;

        // Ограничивающий прямоугольник становится
        // нереальным
        bmin_x = 100000;
        bmax_x = -100000;

```

```

bmin_y = 100000;
bmax_y = -100000;

// Основываясь на расстоянии по оси z, затемняем
// корабль, нормализуем расстояние от 0 до max_z,
// затем масштабируем (чем ближе, тем ярче)
USHORT rgb_tie_color =
    RGB16Bit(0,(31-31*(ties[index].z/(4*FAR_Z))),0);

// Каждый корабль состоит из ряда ребер
for (int edge=0; edge < NUM_TIE_EDGES; edge++)
{
    POINT3D p1_per, p2_per;
    // Шаг 1: аксонометрическое преобразование
    // каждой конечной точки. Заметим, что
    // трансляция каждой точки в положение боевого
    // корабля, который является моделью,
    // выполняется относительно положения корабля
    p1_per.x =
        VIEW_DISTANCE*(ties[index].x+
        tie_vlist[tie_shape[edge].v1].x)/
        (tie_vlist[tie_shape[edge].v1].z+
        ties[index].z);

    p1_per.y = VIEW_DISTANCE*(ties[index].y+
        tie_vlist[tie_shape[edge].v1].y)/
        (tie_vlist[tie_shape[edge].v1].z+
        ties[index].z);

    p2_per.x = VIEW_DISTANCE*(ties[index].x+
        tie_vlist[tie_shape[edge].v2].x)/
        (tie_vlist[tie_shape[edge].v2].z+
        ties[index].z);

    p2_per.y = VIEW_DISTANCE*(ties[index].y+
        tie_vlist[tie_shape[edge].v2].y)/
        (tie_vlist[tie_shape[edge].v2].z+
        ties[index].z);

    // Шаг 2: вычисляем экранные координаты
    int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
    int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
    int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
    int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

    // Шаг 3: рисуем ребро
    Draw_Clip_Line16(p1_screen_x, p1_screen_y,
        p2_screen_x, p2_screen_y,
        rgb_tie_color, back_buffer,
        back_pitch);

    // Обновляем ограничивающий прямоугольник
    // следующим ребром
}

```

```

int min_x = min(p1_screen_x, p2_screen_x);
int max_x = max(p1_screen_x, p2_screen_x);

int min_y = min(p1_screen_y, p2_screen_y);
int max_y = max(p1_screen_y, p2_screen_y);

bmin_x = min(bmin_x, min_x);
bmin_y = min(bmin_y, min_y);

bmax_x = max(bmax_x, max_x);
bmax_y = max(bmax_y, max_y);

} // for edge

// Проверяем, был ли этот корабль поражен лазерами
if (cannon_state==1)
{
    // Простой тест экранных координат
    // ограничивающего прямоугольника, который
    // содержит лазерную цель
    if (target_x_screen > bmin_x &&
        target_x_screen < bmax_x &&
        target_y_screen > bmin_y &&
        target_y_screen < bmax_y)
    {
        // Этот корабль мертв!
        Start_Explosion(index);

        // Запускаем звук
        DSound_Play(explosion_id );

        // Увеличиваем счет
        score+=ties[index].z;

        // Добавляем одно очко
        hits++;

        // Последняя переустановка
        // параметров этого корабля
        Init_Tie(index);
    }
} // if

} // if

} // for index

} // Draw_Ties

////////////////////////////////////

int Game_Main(void *parms)
{

```

```
// Рабочий блок игры, он будет постоянно вызываться
// в режиме реального времени. Подобен функции main()
// в языке C
```

```
int index;
```

```
// Запускаем часы
Start_Clock();
```

```
// Очищаем поверхность рисования
DDraw_Fill_Surface(lpddsback, 0);
```

```
// Получаем ввод с клавиатуры и других устройств
DInput_Read_Keyboard();
```

```
// Блок логики игры...
```

```
if (game_state==GAME_RUNNING)
{
    // Перемещаем перекрестие прицела
    if (keyboard_state[DIK_RIGHT])
    {
        // Перемещаем перекрестие прицела вправо
        cross_x+=CROSS_VEL;

        if (cross_x > WINDOW_WIDTH/2)
            cross_x = WINDOW_WIDTH/2;

    } // if
    if (keyboard_state[DIK_LEFT])
    {
        // Перемещаем перекрестие прицела влево
        cross_x-=CROSS_VEL;

        if (cross_x < -WINDOW_WIDTH/2)
            cross_x = -WINDOW_WIDTH/2;
    } // if

    if (keyboard_state[DIK_DOWN])
    {
        // Перемещаем перекрестие прицела вниз
        cross_y+=CROSS_VEL;

        if (cross_y > WINDOW_HEIGHT/2)
            cross_y = WINDOW_HEIGHT/2;
    } // if

    if (keyboard_state[DIK_UP])
    {
        // Перемещаем перекрестие прицела вверх
        cross_y-=CROSS_VEL;

        if (cross_y < -WINDOW_HEIGHT/2)
            cross_y = -WINDOW_HEIGHT/2;
    } // if
}
```

```

        cross_y - WINDOW_HEIGHT/2;
    } // if

    // Управление скоростью корабля
    if (keyboard_state[DIK_A])
        player_z_vel++;
    else
        if (keyboard_state[DIK_S])
            player_z_vel--;

    // Проверяем, стреляет ли игрок из лазерной пушки
    if (keyboard_state[DIK_SPACE] && cannon_state==0)
    {
        // Стреляем из пушки
        cannon_state = 1;
        cannon_count = 0;

        // Сохраняем последнее положение прицела
        target_x_screen = cross_x_screen;
        target_y_screen = cross_y_screen;

        // Выводим звук
        DSound_Play(laser_id);
    } // if
}

// Процесс работы пушки - простой конечный автомат:
// готовность-выстрел-охлаждение

// Фаза выстрела
if (cannon_state == 1)
    if (++cannon_count > 15)
        cannon_state = 2;

-// фаза охлаждения
if (cannon_state == 2)
    if (++cannon_count > 20)
        cannon_state = 0;

// Перемещаем звездное поле
Move_Starfield();

// Перемещаем и выполняем блок искусственного
// интеллекта для кораблей
Process_Ties();

// Обработка взрывов
Process_Explosions();

// Закрываем задний буфер
DDraw_Lock_Back_Surface();

```

```

// Рисуем звездное поле
Draw_Starfield();

// Рисуем боевые корабли
Draw_Ties();

// Рисуем взрывы
Draw_Explosions();

// Рисуем перекрестие прицела
// Вначале вычисляем экранные координаты перекрестия
// прицела. Обратите внимание на знак по оси y
cross_x_screen = WINDOW_WIDTH/2 + cross_x;
cross_y_screen = WINDOW_HEIGHT/2 - cross_y;

// Рисуем перекрестие прицела в экранных координатах
Draw_Clip_Line16(cross_x_screen-16,cross_y_screen,
                 cross_x_screen+16,cross_y_screen,
                 rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen,cross_y_screen-16,
                 cross_x_screen,cross_y_screen+16,
                 rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen-16,cross_y_screen-4,
                 cross_x_screen-16,cross_y_screen+4,
                 rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen+16,cross_y_screen-4,
                 cross_x_screen+16,cross_y_screen+4,
                 rgb_red,back_buffer,back_lpitch);

// Рисуем лазерные лучи
if (cannon_state == 1)
{
    if ((rand()%2 == 1))
    {
        // Правый луч
        Draw_Clip_Line16(WINDOW_WIDTH-1,WINDOW_HEIGHT-1,
                        -4+rand()%8+target_x_screen,
                        -4+rand()%8+target_y_screen,
                        RGB16Bit(0,0,rand()),
                        back_buffer,back_lpitch);
    }
    else
    {
        // Левый луч
        Draw_Clip_Line16(0, WINDOW_HEIGHT-1,
                        -4+rand()%8+target_x_screen,
                        -4+rand()%8+target_y_screen,
                        RGB16Bit(0,0,rand()),
                        back_buffer,back_lpitch);
    }
}

```

```

    } // if

} // if

// Визуализация завершена, открываем
// поверхность заднего буфера
DDraw_Unlock_Back_Surface();

// Выводим текстовую информацию
sprintf (buffer, "Score %d Kills %d Escaped %d",
        score, hits, misses);
Draw_Text_GDI(buffer, 0,0,RGB(0,255,0), lpddsback);

if (game_state==GAME_OVER)
    Draw_Text_GDI("G A M E O V E R", 320-8*10,240,
        RGB(255,255,255), lpddsback);

// Проверяем, закончилась ли музыка,
// если да - перезапускаем
if (DMusic_Status_MIDI(main_track_id)==MIDI_STOPPED)
    DMusic_Play(main_track_id);

// Меняем поверхности
DDraw_Flip();

//Синхронизация 30 кадров в секунду
Wait_Clock(30);

// Проверяем переключатель состояния игры
if (misses > 100)
    game_state = GAME_OVER;

// Проверяем, выходит ли пользователь из игры
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
{
    PostMessage(main_window_handle, WM_DESTROY,0,0);
} // if

// Возвращаем код завершения
return (1);
} // Game_Main

```

////////////////////////////////////

Очень немного для трехмерной игры, правда? Это **реальная** трехмерная игра Win32/DirectX.

Прежде чем мы **займемся** анализом кода, я хочу, чтобы вы сами скомпилировали его. Я запрещаю вам **двигаться** дальше, пока компиляция не завершится успехом! Надеюсь, я понятно выразился. Итак, займитесь настройкой компилятора в соответствии с описанными выше инструкциями для создания Win32 **.EXE-приложений**, указывая пути поиска и создавая списки связывания, настроенные для DirectX. Затем, когда проект будет готов, подключите исходные файлы **T3DLIB1.CPP, T3DLIB2.CPP, T3DLIB3.CPP, RAIDERS3D.CPP**.

Конечно же, заголовочные файлы `T3DLIB1.H`, `T3DLIB2.H`, `T3DLIB3.H` должны быть в рабочем каталоге компилятора. И наконец, необходимо быть абсолютно уверенным, что вы включили в проект `.LIB`-файлы DirectX вместе с `.CPP`-файлами или включили их в список связывания. Вам необходимы лишь следующие `.LIB`-файлы DirectX: `DDRAW.LIB`, `DSOUND.LIB`, `DINPUT.LIB`, `DINPUT8.LIB`.

Вы можете назвать `.EXE`-файл как угодно — возможно, `TEST.EXE` или `RAIDERS3D_TEST.EXE` — однако не идите дальше, пока вы не сможете его скомпилировать.

## Цикл событий

Главная точка входа для всех программ Windows — функция `WinMain()`, точно так же, как `main()` — главная точка входа для программ DOS/UNIX. В любом случае `WinMain()` создает окно для `Raiders3D` и затем входит прямо в цикл событий. `WinMain()` начинается с создания и регистрации класса Windows. Затем создается окно игры, после чего производится вызов функции `Game_Init()`, которая выполняет инициализацию игры. После завершения инициализации выполняется вход в стандартный цикл событий Windows, который считывает сообщения. Если сообщение найдено, вызывается процедура Windows `WinProc`, которая обрабатывает его. В противном случае вызывается функция игры `Game_Main()`. Именно здесь происходит реальное действие игры.

### НА ЗАМЕТКУ

Читатели предыдущей книги могут заметить, что в разделе инициализации функции `WinMain()` появился дополнительный код для обработки оконной графики и изменения размера окна. Эта возможность, а также поддержка 16-битового цвета являются частью новой версии игрового процессора `T3DLIB`. Тем не менее, большая часть кода в этой книге по-прежнему поддерживает 8-битовую графику, поскольку в общем случае скорость 16-битовых программ все еще слишком мала.

При желании вы можете войти в бесконечный цикл `Game_Main()` и никогда больше не возвращаться в основной цикл событий `WinMain()`, но это было бы плохо, поскольку в этом случае Windows не будет получать сообщения. Нам нужно рассчитать и вывести один кадр анимации, а затем вернуться в `WinMainQ`. При этом Windows продолжит работу и будет обрабатывать сообщения. Этот процесс показан на рис. 1.17.

## Внутренняя логика игры

После выполнения блока игровой логики в функции `Game_Main()` производится визуализация изображения во внеэкранную рабочую область (двойной буфер, или на жаргоне DirectX "задний буфер" (back buffer)). Завершающим этапом является вывод изображения на экран в конце цикла с помощью вызова `DDraw_Flip()`, что создает иллюзию анимации. Игровой цикл состоит из стандартных разделов, определенных ранее в элементах двумерных или трехмерных игр. Теперь я хочу сосредоточиться на 3D-графике.

Логика искусственного интеллекта врага очень проста. Вражеский корабль создается в случайной точке трехмерного пространства на расстоянии, превышающем видимость. Рисунок 1.18 показывает пространство вселенной `Raiders3D`. Как видно из рисунка, камера или наблюдатель расположены в точке на отрицательной оси  $z$  с координатами  $(0,0,-zd)$ , где  $zd$  — расстояние от наблюдателя до виртуального окна, куда проецируется изображение. Здесь используется левая система координат (положительная ось  $z$  направлена в экран).

После того как вражеский корабль создан, он следует по заданному вектору траектории, пересекающей поле зрения игрока, т.е. он идет более или менее встречным курсом. Вектор и начальное положение корабля генерируются функцией `Init_Tie()`. Вашей целью как игрока является прицелиться во врага и выстрелить.



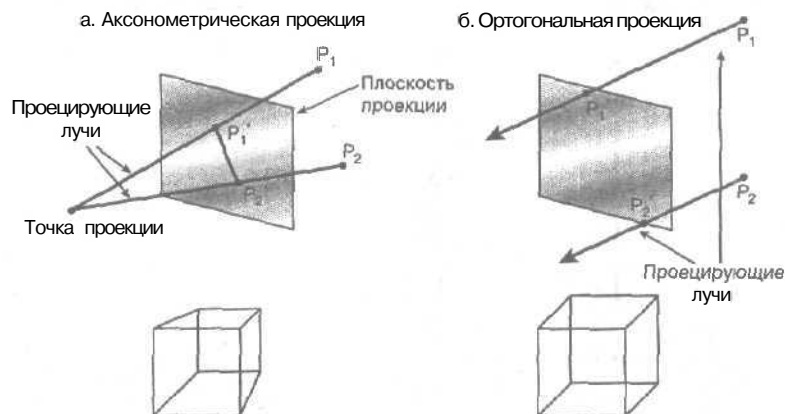


Рис. 1.19. Аксонометрическая и ортогональная аксонометрическая проекции

## Трехмерные проекции

Ортогональная проекция хорошо подходит для технических чертежей и изображений, где аксонометрическое искажение нежелательно. Математика ортогональной проекции очень проста — по сути, происходит простое изменение координаты  $z$  каждой точки в соответствии со следующим уравнением.

### Уравнение 1.1. Ортогональная проекция

Для точки с координатами  $(x, y, z)$

$$x_{\text{ortho}} = x,$$

$$y_{\text{ortho}} = y.$$

Аксонометрическая проекция немного сложнее, и на данном этапе я не хочу слишком вдаваться в ее объяснение. В целом, для получения двумерной проекции на экране с координатами  $(x_{\text{per}}, y_{\text{per}})$  нам необходимо учитывать координату  $z$ , а также расстояние от наблюдателя. Математика аксонометрической проекции показана на рис. 1.20. Она основана на вычислении координат подобных треугольников.

### Уравнение 1.2. Аксонометрическая проекция

Для точки с координатами  $(x, y, z)$  с расстоянием до наблюдателя  $zd$

$$x_{\text{per}} = \frac{zd \cdot x}{z},$$

$$y_{\text{per}} = \frac{zd \cdot y}{z}.$$

С помощью такого простого уравнения объекты, состоящие из многоугольников, в трехмерном пространстве можно перемещать так же, как и двумерные объекты. Применяя к объектам аксонометрическое преобразование до выполнения визуализации, мы достигнем корректный вид и перемещение в трехмерном пространстве. Конечно, здесь есть несколько своих тонкостей, но сейчас они не важны. Все, что нужно знать, — это то, что трех-

мерные объекты проецируются на двумерное поле зрения (экран) по законам аксонометрической проекции.

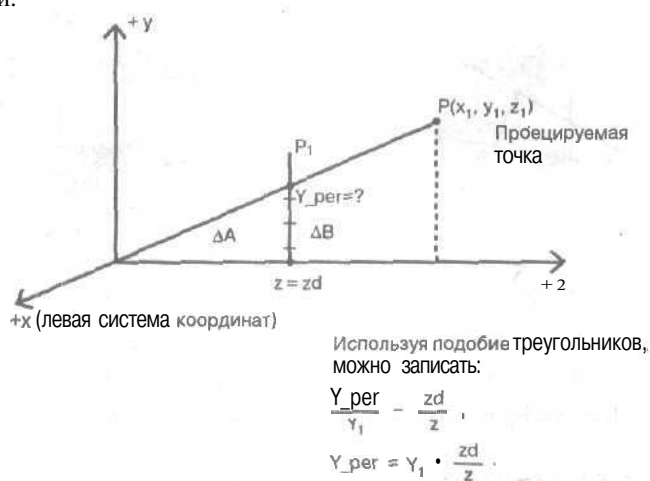
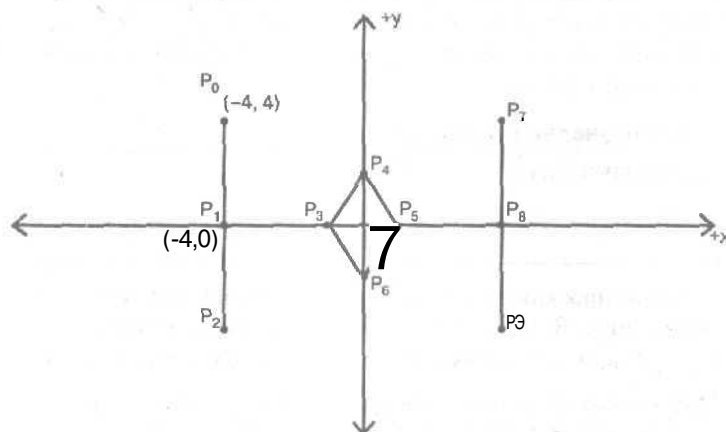


Рис. 1.20. Аксонометрическое преобразование



Список вершин  $P_0 \dots P_9$

$P_0$	$(-4, 4)$
$P_1$	$(-4, 0)$
$P_2$	$(-4, -4)$
$P_3$	$(-1, 0)$
$P_4$	$(0, 2)$
$P_5$	$(1, 0)$
$P_6$	$(0, -2)$
$P_7$	$(4, 4)$
$P_8$	$(4, 0)$
$P_9$	$(4, -4)$

Список ребер

Ребро 0	$P_0-P_2$
Ребро 1	$P_1-P_3$
Ребро 2	$P_3-P_4$
Ребро 3	$P_4-P_5$
Ребро 4	$P_5-P_6$
Ребро 5	$P_6-P_3$
Ребро 6	$P_5-P_8$
Ребро 7	$P_7-P_9$

Рис. 1.21. Каркасная модель боевого корабля

Итак, мы преобразуем каждый вражеский корабль с помощью аксонометрической проекции и выполняем его визуализацию на экран. Рис. 1.21 показывает каркасную мо-

дель вражеского корабля, которая используется в качестве основы для визуализации. Таблица положений каждого корабля хранится в следующем массиве.

```
// Боевой корабль
typedef struct TIE_TYP
{
    int state; // Состояние корабля: 0=мертв, 1=жив
    float x, y, z; // Координаты корабля
    float xv, yv, zv; // Скорость корабля
} TIE, *TIE_PTR;
```

Когда приходит время рисовать вражеские корабли, используются данные в виде структуры TIE и мы получаем более-менее реалистичное трехмерное изображение движущегося объекта.

Вы также заметите, что корабли по мере приближения становятся ярче. Этот эффект легко реализовать. Главный принцип состоит в том, что ось z используется в качестве масштабирующего коэффициента при изменении яркости корабля.

## Звездное поле

Звездное поле — это не что иное, как совокупность одиночных точек, генерируемых некоторым источником в пространстве. Кроме того, после ухода из поля зрения игрока они появляются заново. Точки подвергаются визуализации как полноценные 3D-объекты в соответствии с перспективным преобразованием. Однако их размер — 1x1x1 пиксель, поэтому это "настоящие" точки и всегда выглядят как одиночные пиксели.

## Лазерные пушки и обнаружение попаданий

Лазерные пушки, из которых стреляет игрок, — это не что иное, как двумерные линии, выходящие из углов экрана и сходящиеся на перекрестии прицела. Обнаружение попаданий выполняется путем наблюдения за двумерными проекциями кораблей на поле зрения и проверкой того, попадают ли лазерные лучи в ограничивающий прямоугольник каждой проекции. Это процесс схематически показан на рисунке 1.22.

Этот алгоритм работает, поскольку лазерные лучи распространяются со скоростью света. Не имеет значения, находится ли цель на расстоянии 10 метров или 10000 километров, — если вы прицеливаетесь и лазерный луч пересекает проекцию трехмерного объекта, произойдет попадание.

## Взрывы

Взрывы в этой игре очень впечатляют размером кода. При попадании лазерного луча в корабль, линии, образующие 3D-модель корабля, копируются во вторичную структуру данных. Затем линии беспорядочно расходятся в трехмерном пространстве, что напоминает обломки корабля. Движение линий происходит несколько секунд, после чего взрыв прекращается. Эффект очень реалистичен и реализован меньше чем 100 строками кода.

## Как играть в Raiders3D

Чтобы запустить игру — просто сделайте щелчок мышью на файле RAIDERS3D.EXE на компакт-диске, и программа тотчас запустится. Клавиши управления:

- клавиши управления курсором — перемещение перекрестия прицела;
- пробел — огонь из лазерных пушек;
- Esc — выход из игры.

Игра использует DirectDraw, DirectInput, DirectSound и DirectMusic, поэтому убедитесь, что в системе установлен DirectX.

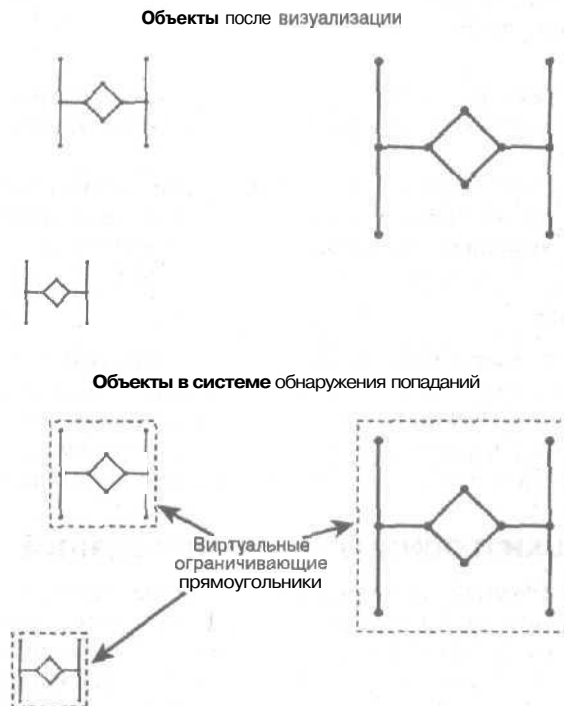


Рис. 1.22. Использование двумерных ограничивающих прямоугольников для обнаружения попаданий

## Резюме

Эта глава представляет собой лишь начало введения в программирование трехмерных игр. Самым важным здесь является компилирование DirectX-программ. Кроме того, мы рассматриваем основы программирования игр, игровой цикл, основы аксонометрического преобразования 3D-объектов, а также использование компилятора. Здесь вкратце рассмотрена библиотека T3DLIB, разработанная в предыдущей книге *Программирование игр для Windows. Советы профессионала*. Она позволит нам сосредоточиться на программировании трехмерных игр, а не на создании поверхностей, изучении интерфейса и загрузке звука. Теперь, когда вы уже умеете компилировать, поэкспериментируйте с игрой — добавьте противников, астероиды, что-нибудь еще — и приступайте к чтению новой главы.

# ГЛАВА 2

## Краткий курс Windows и DirectX

### В этой главе...

• Модель программирования Win32	80
• Необходимый минимум знаний по программированию для Windows	81
• Базовое приложение для Windows	87
• Краткий курс DirectX и COM	104
• Краткое введение в COM	108

Основная цель этой главы — изложение краткого курса программирования для Windows и DirectX. Предполагается, что вы **уже** знакомы с основами программирования для Win32 и **DirectX**, однако я попытаюсь так построить изложение, что даже если вы и **незнакомы** с этим материалом, то все равно сможете просто использовать созданный мной API и сфокусироваться на аспектах трехмерного программирования. Для тех кто не имел удовольствия познакомиться с программированием для **Win32/DirectX**, я позволю себе дать основные сведения по этой теме. Вот что мы рассмотрим в этой главе:

- программирование для Windows;
- циклы событий;
- написание простой Windows-оболочки;
- основы DirectX;
- модель составных объектов.

## Модель программирования Win32

Windows — это многозадачная/многопоточная операционная система. Вместе с тем, она управляется событиями. В отличие от большинства DOS-программ (собственно, всех DOS-программ), большинство Windows-программ находится в состоянии ожидания действий пользователя (являющихся событиями), после чего Windows реагирует на событие и выполняет определенные действия. На рис. 2.1 этот процесс показан графически. Как видно из рисунка, оконные приложения посылают Windows свои события или сообщения для обработки. Определенная обработка выполняется самой Windows, однако большинство сообщений или событий передаются для обработки приложениям.

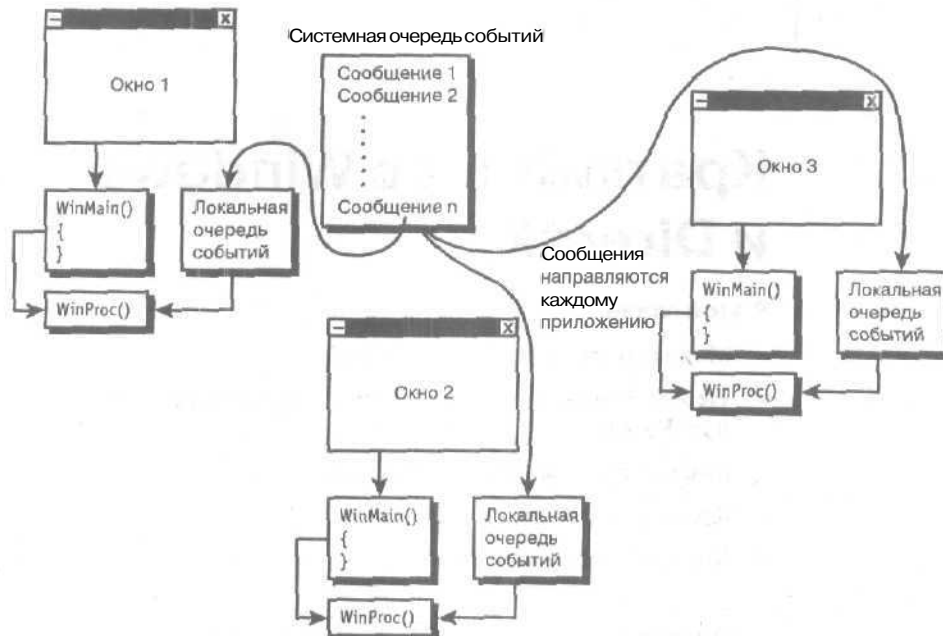


Рис. 2.1. Обработка событий в Windows

Могут вас обрадовать: в большинстве случаев вам не придется заботиться о других работающих приложениях — Windows все сделает за вас. Все, что от вас потребуется, — это беспокоиться о собственном приложении и обработке поступающих для него сообщений. Это было не совсем так в Windows 3.0/3.1. Эти версии Windows не были полностью многозадачными, и каждое приложение должно было само давать сигнал операционной системе о возможности переключения на другое приложение. Это создавало ощущение замедления работы приложений в этих версиях Windows. Если какое-то приложение тормозило систему, другие программы ничего не могли с этим поделать. В Windows 9x/Me/2000/XP/NT ситуация кардинально изменилась. Система при необходимости сама переключает приложения, причем так быстро, что это невозможно заметить!

### НА ЗАМЕТКУ

Теперь вы знаете все, что нужно, о принципах работы операционной системы. К счастью, на сегодняшний день Windows настолько хорошо приспособлена для написания игр, что вам не нужно беспокоиться о распределении ресурсов, планировании процессов и тому подобных вещах. Все, о чем вы должны беспокоиться, — это игровой код и чувство предельных возможностей компьютера.

# Минимальный курс программирования для Windows

Теперь, когда вы уже имеете общее представление об операционной системе Windows, некоторых ее свойствах и базовых принципах, самое время приступить к изложению краткого курса программирования для Windows на примере нашей первой программы для Windows.

Изучение любого нового языка программирования принято начинать с написания программы "Hello World". Мы поступим так же. Ниже приведен листинг стандартной DOS-версии программы "Hello World".

```
// DEMOII2_1.CPP — Стандартная версия программы hello world
#include <stdio.h>
```

```
// Основная точка входа для всех стандартных DOS/консольных
// программ
void main(void)
{
    printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // main
```

Теперь посмотрим, как это делается в Windows.

НА ЗАМЕТКУ

Кстати, если вы компилируете DEMOII2\_1.CPP в компиляторе Visual C++ или Borland, вы можете создать так называемое консольное приложение. Оно напоминает DOS-приложение, однако это 32-битовое приложение. Это приложение работает только в текстовом режиме, но прекрасно подходит для тестирования идей и алгоритмов.

## Все начинается с WinMain()

Выполнение всех программ для Windows начинается с функции WinMain(). Это эквивалент функции main() в случае платформ DOS и UNIX. Что будет делать функция WinMain() — целиком зависит от вас. При желании можно создать окно, начать обработку событий, вывод изображений на экран и т.д. С другой стороны, можно просто вызвать одну из сотен (или тысяч) Win32 API-функций. Именно это мы и будем делать дальше.

В качестве примера я хочу вывести на экран некий текст в маленьком окне сообщений. Для этого существует специальная функция Win32 API — MessageBox(). Ниже приводится листинг нормальной, корректно компилируемой программы для Windows, которая создает и выводит на экран окно сообщения, которое можно двигать и при желании закрыть.

```
// DEMOII2_2.CPP - Простое окно сообщений
#define WIN32_LEAN_AND_MEAN // MFC не используются

#include <windows.h> // Основные заголовочные файлы
#include <windowsx.h>

// Главная точка входа для всех приложений Windows
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
```

```

{
// Вызов функции API с нулевым дескриптором родителя
MessageBox(NULL, "THERE CAN BE ONLY ONE!!!",
    "MY FIRST WINDOWS PROGRAM",
    MB_OK | MB_ICONEXCLAMATION);

// Выход из программы
return(0);

} // WinMain

```

Для компиляции программы выполните следующие шаги.

1. Создайте новый проект **.EXE-приложения Win32** и включите в него файл **DEMOI2\_2.CPP** из каталога **T3DIICHP02\** на компакт-диске.
2. Выполните компиляцию и компоновку программы.
3. Запустите ее (или готовую версию **DEMOI2\_2.EXE** с компакт-диска).

Вы думали, что в **Windows-программе** сотни строк кода? Как бы то ни было, при компиляции и запуске программы вы должны увидеть нечто наподобие рис. 2.2.



Рис. 2.2. Копия экрана **DEMOI2\_2.EXE**

Теперь, когда мы создали полноценную **Windows-программу**, давайте разберем ее строка за строкой и посмотрим, что в ней происходит.

Самая первая строка кода выглядит следующим образом.

```

#define WIN32_LEAN_AND_MEAN

```

Она заслуживает некоторого пояснения. Есть два пути написания программ для **Windows**: с помощью **MFC** (**Microsoft Foundation Classes**) или **SDK** (**Software Development Kit** — набор средств для разработки программного обеспечения). **MFC** — значительно более сложный инструмент, основанный полностью на **C++** и классах. Возможности **MFC** в десятки раз превышают ваши потребности в области программирования игр. С другой стороны, **SDK** — это гибкий инструмент, который можно изучить в течение недели или двух (по крайней мере, его основы). Он использует обычный **C**. Поэтому **SDK** — это именно то, чем мы будем пользоваться в этой книге.

Возвращаясь к объяснению. Макроопределение **WIN32\_LEAN\_AND\_MEAN** дает компилятору команду не подключать внешние **MFC** классы.

Затем включаются следующие заголовочные файлы.

```

#include <windows.h>
#include <windowsx.h>

```

Первое включение (**windows.h**) на самом деле включает все заголовочные файлы **Windows**, избавляя вас от необходимости вручную включать десятки заголовочных файлов.

Второе включение (**windowsx.h**) — это включение заголовочных файлов с наборами макросов и констант, которые значительно облегчают программирование для **Windows**.

А вот и самая важная часть — главная входная точка всех приложений **Windows**, **WinMain()**.

```
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow);
```

Прежде всего, вы должны обратить внимание на этот странный декларатор объявления — WINAPI. Этот эквивалент декларатора PASCAL, который заставляет компилятор передавать параметры слева направо, в отличие от обычного порядка передачи справа налево в принятом по умолчанию соглашении о вызовах CDECL. Однако декларатор PASCAL устарел и его место занял декларатор WINAPI. Вы **обязаны** использовать WINAPI для функции WinMain(), в противном случае загрузочный код будет неправильно передавать параметры в функцию!

Теперь давайте подробно рассмотрим каждый параметр функции.

- **hinstance.** Этот параметр представляет собой дескриптор экземпляра, который Windows генерирует для приложения. *Экземпляры* используются для отслеживания используемых ресурсов. В нашем случае hinstance используется для отслеживания параметров приложения, таких как имя или адрес. При запуске приложения Windows передает ему данный параметр.
- **hprevinstance.** Этот параметр больше не используется, однако в предыдущих версиях Windows он использовался для отслеживания предыдущего экземпляра приложения, иными словами, экземпляра приложения, которое запустило данное.
- **lpcmdline.** Строка с завершающим нулевым символом; аналог командной строки стандартной C/C++ функции main() — с тем отличием, что здесь нет отдельного параметра, аналогичного argc, показывающего количество параметров командной строки. Например, если вы создаете Windows-приложение под названием TEST.EXE и запускаете его со следующими параметрами;

```
TEST.EXE one two three
```

то lpcmdline будет содержать следующие данные:

```
lpcmdline = "one two three"
```

Заметим, что само имя .EXE-файла *не* является частью командной строки.

- **ncmdshow.** Этот последний параметр — просто целое число, передаваемое приложению во время запуска и определяющее, как именно должно открываться основное приложение. Таким образом, у пользователя есть небольшая возможность управления тем, как будет запускаться приложение. Конечно, вы как программист можете при желании отказаться от этой возможности. Однако этот параметр создан для случая, когда вы захотите воспользоваться им (вы передаете его ShowWindow(), однако здесь мы уже забегаем вперед). Таблица 2.1 содержит основные значения, которые может принимать параметр ncmdshow.

**Таблица 2.1. Windows-коды для параметра ncmdshow**

Код	Значение кода
SW_SHOWNORMAL	Активизирует и отображает окно. Если окно минимизировано или максимизировано, Windows восстанавливает его до нормального размера и положения. Приложение должно устанавливать этот флаг при первом отображении окна

Код	Значение кода
SW_SHOW	Активизирует окно и отображает его для данных размеров и положения
SW_HIDE	Прячет окно и активизирует другое окно
SW_MAXIMIZE	Максимизирует указанное окно
SW_MINIMIZE	Минимизирует указанное окно и активизирует следующее окно в Z-порядке
SW_RESTORE	Активизирует и отображает окно. Если окно минимизировано или максимизировано, Windows восстанавливает его до первоначальных размеров и положения. Приложение должно устанавливать этот флаг при восстановлении минимизированного окна
SW_SHOWMAXIMIZED	Активизирует окно и максимизирует его
SW_SHOWMINIMIZED	Активизирует окно и минимизирует его
SW_SHOWMINNOACTIVE	Минимизирует окно. При этом активное окно остается активным
SW_SHOWNORMAL	Отображает окно в его текущем состоянии. Активное окно остается активным
SW_SHOWNORMAL	Отображает окно с его последними размерами и в последнем положении. Активное окно остается активным

Как видно из табл. 2.1, есть множество кодов для `ncmdshow` (многие из которых в данный момент не имеют смысла для нас). На практике большинство из них никогда не передаются в `cmdshow`; вы будете использовать их с другой функцией — `ShowWindow()`, которая отображает окно после его создания. Однако мы рассмотрим этот вопрос несколько позднее. Сейчас я хочу обратить ваше внимание на то, что в Windows есть множество опций, флагов и т.п., которые вы никогда не будете использовать, но все же они есть. Это аналогично возможностям видеомагнитофона: больше — это всегда лучше, но при этом вам необязательно использовать их все, если вы этого не хотите. Именно так построена Windows. Чтобы сделать каждого пользователя счастливым, в ней предусмотрено множество возможностей. Но фактически 99% времени мы будем использовать только `SW_SHOW`, `SW_SHOWNORMAL` и `SW_HIDE`.

Теперь давайте поговорим о вызове функции `MessageBox()` внутри `WinMain()`. Этот вызов делает за нас всю работу. `MessageBox()` — это функция Win32 API, которая делает для нас некоторые полезные вещи и тем самым избавляет нас от необходимости делать их самим. `MessageBox()` используется для отображения сообщений с различными пиктограммами и одной или двумя кнопками. Как видите, отображение простых сообщений в приложениях Windows — это тривиальная задача благодаря функции, написанной специально для того, чтобы программист каждый раз мог экономить полчаса времени на написание подобной программы.

`MessageBoxQ` делает не очень много, но этого достаточно для вывода на экран окна, задания вопроса и ожидания ввода пользователя. Ниже дается прототип `MessageBox()`.

```
int MessageBox(
    HWND hwnd,    // Дескриптор окна пользователя
    LPCTSTR lptext, // Текст в окне сообщения
    LPCTSTR lpcaption, // Заголовок окна сообщения
    UINT utype);  // Стил ь окна сообщения
```

- `hwnd` — дескриптор окна, к которому вы хотите присоединить окно **сообщения**. Пока что мы не рассматривали дескриптор окна, поэтому считайте этот параметр просто **родителем** окна **сообщений**. В случае `DEMO12_2.CPP` мы устанавливаем его равным нулю. Это означает, что мы используем в качестве родительского окна рабочий стол Windows.
- `lpText` — строка с завершающим нулем, содержащая отображаемый текст.
- `lpCaption` — строка с завершающим нулем, содержащая заголовок диалогового окна сообщений.
- `uType` — пожалуй, единственный интересный параметр из всего множества параметров функции. Он определяет тип отображаемого окна сообщений. В табл. 2.2 перечислены возможные значения этого параметра.

**Таблица 2.2. Типы окон `MessageBox()`**

Флаг	Значение флага
<b>Общий вид окна сообщений</b>	
<code>MB_OK</code>	Окно <b>сообщений</b> содержит одну кнопку ОК. Это режим по умолчанию
<code>MB_OKCANCEL</code>	Окно сообщений содержит две кнопки: <b>ОК</b> и Cancel
<code>MB_RETRYCANCEL</code>	Окно сообщений содержит две кнопки: Cancel и Retry
<code>MB_YESNO</code>	Окно <b>сообщений</b> содержит две кнопки: Yes и No
<code>MB_YESNOCANCEL</code>	Окно сообщений содержит три кнопки: Yes, No и Cancel
<code>MB_ABORTRETRYIGNORE</code>	Окно сообщений содержит три кнопки: Abort, Retry и Ignore
<b>Пиктограмма в окне сообщения</b>	
<code>MB_ICONEXCLAMATION</code>	В окне сообщений появляется пиктограмма с восклицательным знаком
<code>MB_ICONINFORMATION</code>	В окне сообщений появляется буква i в окружности.
<code>MB_ICONQUESTION</code>	В окне сообщений появляется пиктограмма с вопросительным знаком
<code>MB_ICONSTOP</code>	В окне сообщений появляется пиктограмма со знаком "стоп"
<b>Кнопка, используемая по умолчанию</b>	
<code>MB_DEFBUTTONn</code>	n — это число (1...4), обозначающее номер кнопки (слева направо), используемой по умолчанию

*Примечание.* Имеются и другие флаги уровня операционной системы, однако для нас они не представляют интереса. При необходимости вы всегда сможете узнать о них в справочной системе Win32 SDK.

Вы можете применять побитовое ИЛИ к значениям из таблицы 2.2 для создания требуемого окна сообщений. Обычно выбирают только один флаг из каждой группы.

И конечно, как и подавляющее большинство функций Win32 API, `MessageBox()` возвращает значение, которое позволяет узнать, что произошло. В нашем случае это никому не нужно, но в общем случае может возникнуть необходимость узнать возвращаемое значение, например, если окно сообщения содержит вопрос Yes/No и т.п. Возможные возвращаемые значения перечислены в табл. 2.3.

**Таблица 2.3. Возвращаемые значения функции MessageBox()**

<i>Значение</i>	<i>Его смысл</i>
IDABORT	Была выбрана кнопка Abort
IDCANCEL	Была выбрана кнопка Cancel
IDIGNORE	Была выбрана кнопка Ignore
IDNO	Была выбрана кнопка No
ШОК	Была выбрана кнопка OK
IDRETRY	Была выбрана кнопка Retry
IDYES	Была выбрана кнопка Yes

Теперь я хочу, чтобы вы набрались терпения, поскольку вам придется вносить изменения в программу и компилировать ее различными способами. Попробуйте менять различные опции **компилятора** — такие как оптимизация, генерация кода и т.д. **Затем** попробуйте пропустить программу через отладчик и во всем этом разобраться.

Если вы хотите услышать какой-нибудь звук — поэкспериментируйте с функцией `MessageBeep()`. О ней можно узнать в справочной системе Win32 SDK. По простоте использования она напоминает функцию `MessageBox()`.

```
BOOL MessageBeep(UINT utype); // Звук, выводимый компьютером
```

Здесь различным звукам соответствуют константы, показанные в табл. 2.4.

**Таблица 2.4. Тип звука функции MessageBeep()**

<i>Параметр</i>	<i>Значение параметра</i>
MB_ICONASTERISK	Системный звук "звездочка"
MB_ICONEXCLAMATION	Системный звук "восклицание"
MB_ICONHAND	Системный звук "рука"
MB_ICONQUESTION	Системный звук "вопрос"
MB_OK	Системный звук по умолчанию
0xFFFFFFF	Стандартный тоновый сигнал, воспроизводимый встроенным динамиком компьютера

*Примечание.* Если у вас установлены темы **MS-Plus**, то вы получите очень интересные результаты.

Ну что, теперь вы видите, насколько силен Win32 API? В нем буквально сотни интересных функций. Конечно, это не самые лучшие функции в мире, но для **общего** пользования, для системы ввода-вывода и графического интерфейса пользователя Win32 API весьма удобен.

Теперь давайте резюмируем все, что мы знаем на данный момент о программировании для Windows. Во-первых, Windows — это многозадачная/многопоточная система, а поэтому в ней можно одновременно запускать несколько приложений. Однако что нас действительно интересует — это то, что Windows управляется событиями. Это означает, что мы должны обрабатывать события (о чем мы на данный момент не имеем представления) и реагировать на них. Наконец, все программы Windows начинаются с функции `WinMain()`, которая отличается от обычной DOS-функции `main()` наличием нескольких дополнительных параметров, на что есть веские причины.

Учитывая все сказанное, самое время написать базовое приложение для Windows, которое будет основой шаблона игрового процессора, которым мы займемся позднее в этой главе.

## Базовое приложение для Windows

Поскольку цель этой книги — написание трехмерных игр, которые будут работать в операционной системе Windows, нет необходимости много знать о программировании для Windows. Фактически все, что нужно знать, — это базовую структуру программы для Windows, которая открывает окно, обрабатывает сообщения и вызывает основной игровой цикл, — и это все. В данном разделе книги моя цель — это прежде всего научить вас создавать простые программы для Windows. Кроме того, мы выполним подготовительную работу для создания игровой "оболочки", которая будет весьма похожа на 32-битовое DOS/UNIX-приложение. Итак, давайте начнем.

Основной момент в любой Windows-программе — это открытие окна. Окно — это всего лишь рабочая область, в которую выводится информация, такая как текст и графика, и с которой может взаимодействовать пользователь. Для создания полностью функциональной программы для Windows необходимо выполнить следующие шаги.

- Создать класс Windows.
- Создать обработчик событий (WinProc).
- Зарегистрировать класс окна в операционной системе Windows.
- Создать окно с созданным классом Windows.
- Создать главный цикл событий, который получает и передает сообщения обработчику событий.

Рассмотрим каждый шаг подробно.

## Класс Windows

Windows — это действительно объектно-ориентированная операционная система. В Windows есть множество концепций и функций, которые берут свое начало из C и C++. К ним относятся и классы окон. Каждое окно, элемент управления, список, диалоговое окно и другие объекты в Windows — это на самом деле окна. Друг от друга они отличаются определяющим их классом. Классы описывают типы окон, которые могут обрабатываться системой Windows.

Есть множество предопределенных классов окон, таких как кнопки, окна списков, окна выбора файлов и т.д. Вы можете создавать и свои собственные классы окон. Вы должны создать, по крайней мере, один класс окна для каждого приложения, которое вы пишете, — иначе ваша программа будет, скажем так, скучноватой. Класс окна можно рассматривать как шаблон, которому должна следовать Windows при выводе вашего окна и при обработке сообщений для него.

Для хранения информации о классе окна есть две структуры данных WNDCLASS и WNDCLASSEX. WNDCLASS старше и, вероятно, вскоре устареет. Поэтому мы будем использовать новую, расширенную версию WNDCLASSEX. Структуры очень похожи, и если вам интересно, то можете посмотреть на определение WNDCLASS в справочной системе Win32. Давайте рассмотрим, как определяется WNDCLASSEX в заголовочных файлах Windows.

```
typedef struct _WNDCLASSEX
{
    UINT cbSize; // Размер данной структуры
    UINT style; // Флаги стиля
    WNDPROC lpfnWndProc; // Указатель на обработчик
    int cbClsExtra; // Доп. информация о классе
    int cbWndExtra; // Доп. информация об окне
    HANDLE hInstance; // Экземпляр приложения
    HICON hIcon; // Основная пиктограмма
    HCURSOR hCursor; // Курсор окна
    HBRUSH hbrBackground; // Кисть для рисования фона окна
    LPCTSTR lpszMenuName; // Имя прикрепляемого меню
    LPCTSTR lpszClassName; // Имя класса
    HICON hIconSm; // Дескриптор пиктограммы
} WNDCLASSEX;
```

Итак, все, что мы делаем, — это **создаем** одну из этих структур, а затем заполняем ее поля.

```
WNDCLASSEX winclass; // Пустой класс окна
```

Теперь давайте рассмотрим, как заполнять каждое поле.

Первое поле `cbSize` очень важно. Это размер самой структуры **WNDCLASSEX**. Можно спросить, а зачем структуре нужно знать, каков ее размер? Это хороший вопрос. Ответ состоит в **том**, что если эта структура передается как указатель, получатель всегда может проверить первое поле (4 байта), чтобы определить предельную длину пакета данных и перейти сразу в его конец. Это **своего** рода предосторожность и небольшая вспомогательная информация, благодаря чему другим функциям не нужно вычислять размер класса во время исполнения программы. Все, что нужно сделать, это **записать**

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

Следующее поле имеет дело с флагами информации о стилях, которые описывают общие свойства окна. Этих флагов **много**, и я не собираюсь показывать их все. Достаточно сказать, что с их помощью можно создать любой тип окна. Хороший практичный набор флагов приведен в табл. 2.5. Для создания нужного типа окна можно использовать эти параметры с оператором побитового ИЛИ.

**Таблица 2.5. Флаги стилей классов окон**

<b>Флаг</b>	<b>Описание</b>
<b>CS_HREDRAW</b>	Если при перемещении или коррекции размера <b>изменяется ширина окна</b> , требуется перерисовка всего окна
<b>CS_VREDRAW</b>	Если при перемещении или коррекции размера <b>изменяется высота окна</b> , требуется перерисовка всего окна
<b>CS_OWNDC</b>	Каждому <b>окну</b> данного класса <b>выделяется свой контекст устройства</b> (более подробно об этом будет сказано позже)
<b>CS_DBLCLKS</b>	При двойном щелчке мышью <b>в тот момент</b> , когда <b>курсор находится в окне</b> , процедуре окна посылается <b>сообщение о двойном щелчке</b>
<b>CS_PARENTDC</b>	Устанавливает область обрезки дочернего окна равной области обрезки родительского окна, так что дочернее окно может изображаться в родительском

Флаг	Описание
CS_SAVEBITS	Сохраняет изображение в окне, так что вам не нужно перерисовывать его каждый раз, когда оно закрывается, перемещается и т.д. Однако для этого требуется много памяти, и обычно такая процедура медленнее, чем если бы вы перерисовывали окно сами
CS_NOCLOSE	Отключает команду <b>Close</b> в системном меню

**Примечание.** Наиболее часто **употребляемые** флаги выделены полужирным шрифтом.

Я привел только те флаги, которые используются в случае перерисовки окна при его перемещении или изменении размеров, а также в случае, когда требуется использовать контекст устройства вместе с возможностью обработки событий при двойном щелчке мыши.

**Контекст устройства** используется при визуализации GDI-графики в окне. Если вы хотите создавать графику, необходимо запрашивать контекст устройства для конкретного интересующего вас окна. Если мы определяем класс окна, в котором **есть** собственный контекст устройства, с **помощью** флага **CS\_OWNDC** мы можем сэкономить время за счет отказа от постоянного запрашивания контекста. Вот пример **того**, как с помощью выбора стиля можно приспособить окно под наши потребности.

```
winclass.style = CS_VREDRAW | CS_HREDRAW |
                CS_OWNDC | CS_DBLCLKS;
```

Очередное поле структуры **WNDCLASSEX** — **lpfnWndProc** — представляет собой указатель на функцию-обработчик событий. Это — функция *обратного вызова* (callback). Такие функции достаточно **широко** распространены в Windows и работают следующим образом. Когда происходит некоторое событие, Windows уведомляет вас о нем вызовом предоставляемой **вами** функции обратного вызова, в которой и выполняется обработка этого события.

Вы предоставляете функцию обратного вызова классу окна (конечно, функция должна иметь определенный прототип) и, когда происходит событие, Windows вызывает ее для вас. Это схематически показано на рис. 2.3. Эта тема будет подробнее рассмотрена в следующих разделах. А пока просто укажем используемую классом функцию событий.

```
winclass.lpfnWndProc = WinProc; // Функция событий
```



Рис. 2.3. Функция обратного вызова обработчика событий **Windows** в действии

#### НА ЗАМЕТКУ

Если вы незнакомы с указателями на функции, то можно считать, что они в чем-то похожи на виртуальные функции C++. Если вы незнакомы с виртуальными функциями, я попытаюсь объяснить, что это такое :-). Пусть у нас есть две функции, работающие с двумя числами.

```

int Add(int op1, int op2) { return (op1 + op2); }
int Sub(int op1, int op2) { return (op1 - op2); }

Вы хотите иметь возможность вызывать любую из них посредством одного и того же
вызова. Это можно сделать с помощью указателя на функцию следующим образом.

// Определим указатель на функцию, получающую в качестве
// параметров два целых числа и возвращающую целое число.
int (*Math)(int, int);

Теперь вы можете присвоить значение указателю и использовать его для вызова со-
ответствующей функции.

Math = Add;
int result = Math(1,2); // Вызывается Add(1,2)
                      //result=3

Math = Sub;
int result = Math(1,2); // Вызывается Sub(1,2)
                      //result = -1

Красиво, не правда ли?

```

Следующие два поля, `cbClsExtra` и `cbWndExtra`, первоначально были созданы, чтобы позволить Windows сэкономить немного места в классе окна для хранения дополнительной информации об исполняемой программе. Большинство программистов не используют эти поля и просто присваивают им значение 0.

```

winclass.cbClsExtra = 0; // Поля дополнительной информации
winclass.cbWndExtra = 0; // класса и окна

```

Следующее поле — `hInstance`. Это просто дескриптор `hinstance`, передаваемый в функцию `WinMain()` при запуске.

```

winclass.hInstance = hinstance; // Экземпляр приложения

```

Оставшиеся поля относятся к графическим аспектам класса окна, однако прежде чем обсудить их, я хочу сделать небольшой обзор дескрипторов.

Мы постоянно встречаемся с дескрипторами в Windows-программах: дескрипторы изображений, дескрипторы курсоров, дескрипторы чего угодно! Дескрипторы — это всего лишь идентификаторы некоторых внутренних типов Windows. В действительности это просто целые числа, но, так как теоретически Microsoft может изменить это представление, лучше все же использовать внутренние типы. В любом случае вы будете очень часто встречаться с теми или иными дескрипторами. Запомните, что имена дескрипторов начинаются с буквы `h` (handle).

Очередное поле структуры определяет пиктограмму, представляющую ваше приложение. Вы можете загрузить собственную пиктограмму или использовать одну из системных. Для получения дескриптора системной пиктограммы можно воспользоваться функцией `LoadIcon()`.

```

winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);

```

Этот код загружает стандартную пиктограмму приложения — невыразительно, зато очень просто. Если вас интересует функция `LoadIcon()`, обратитесь к справке по Win32 API — там приведен ряд готовых пиктограмм, которые можно использовать в своих приложениях.

Итак, половину полей мы уже прошли. Приступим ко второй половине, которая начинается с поля `hCursor`. Оно похоже на поле `hIcon` в том плане, что также представляет собой дескриптор графического объекта. Однако `hCursor` отличается тем, что это дескрип-

тор курсора, который изображается, когда указатель находится в клиентской области окна. Для получения дескриптора курсора используется функция `LoadCursor()`, которая так же, как и `LoadIcon()`, загружает изображение курсора из ресурсов приложения (о ресурсах речь пойдет несколько позже).

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Если вас интересуют другие возможные стили курсора, обратитесь к справочной системе по Win32 API.

Очередное поле структуры — `hbrBackground`. Когда окно выводится на экран или перерисовывается, `Windows`, как минимум, перерисовывает фон клиентской области окна с использованием предопределенного цвета или, в терминах `Windows`, *кисти* (brush). Следовательно, `hbrBackground` — это дескриптор кисти, используемой для обновления окна. Кисти, перья, цвета — все эти части GDI (Graphics Device Interface — интерфейс графического устройства) более подробно рассматриваются в следующей главе. Сейчас же я просто покажу, каким образом можно запросить базовую системную кисть для закраски окна. Это выполняется посредством функции `GetStockObject()`, как показано в следующей строке типа (обратите внимание на приведение возвращаемого типа к `HBRUSH`).

```
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

Функция `GetStockObject()` выдает дескриптор объекта из семейства кистей, перьев, палитр или шрифтов `Windows`. Она получает один параметр, указывающий, какой именно ресурс следует загрузить. В табл. 2.6 приведен список возможных объектов (только кисти и перья).

**Таблица 2.6. Идентификаторы объектов `GetStockObject()`**

Значение	Описание
<code>BLACK_BRUSH</code>	Черная кисть
<code>WHITE_BRUSH</code>	Белая кисть
<code>GRAY_BRUSH</code>	Серая кисть
<code>LTGRAY_BRUSH</code>	Светло-серая кисть
<code>DKGRAY_BRUSH</code>	Темно-серая кисть
<code>HOLLOW_BRUSH</code>	Пустая кисть
<code>NULL_BRUSH</code>	Нулевая кисть
<code>BLACK_PEN</code>	Черное перо
<code>WHITE_PEN</code>	Белое перо
<code>NULL_PEN</code>	Нулевое перо

В большинстве случаев фоновая кисть окна не имеет значения, поскольку ответственность за вывод на экран берет на себя `DirectX`.

Следующее поле в структуре `WNDCLASS` — это поле меню `lpszMenuName`. Это ASCII-строка имени ресурса меню, заканчивающаяся нулем. Меню загружается и прикрепляется к окну. Поскольку мы не собираемся работать с меню, просто присвоим параметру нулевое значение.

```
winclass.lpszMenuName = NULL; // Имя присваиваемого меню
```

Как уже упоминалось, каждый класс `Windows` представляет отдельный тип окна, которое может создать ваше приложение. `Windows` требуется каким-то образом различать разные классы окон, для чего и служит поле `lpszClassName`. Это завершающаяся нулевым символом строка

с текстовым идентификатором данного класса. Лично я предпочитаю использовать в качестве имен строки типа "WINCLASS1", "WINCLASS2" и т.д. Впрочем, это дело вкуса.

```
winclass.lpszClassName = "WINCLASS1";
```

После данного присвоения вы сможете обращаться к новому классу Windows по его имени.

Последним по порядку, но не по значимости идет поле `hIconSm` — дескриптор малой пиктограммы. Это поле добавлено в структуру `WNDCLASSEX` и в старой структуре `WNDCLASS` отсутствовало. Оно представляет собой дескриптор пиктограммы, которая выводится в полосе заголовка вашего окна и на панели задач Windows. Обычно здесь загружается пользовательская пиктограмма, но сейчас мы просто воспользуемся одной из стандартных пиктограмм Windows с помощью функции `LoadIcon()`.

```
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

Вот и все. Итак, вот как выглядит определение класса полностью.

```
WNDCLASSEX winClass;
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_VREDRAW | CS_HREDRAW |
    CS_OWNDC | CS_DBLCLKICKS;
winclass.lpfnWndProc = WinProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hInstance;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = "WINCLASS1";
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

Конечно, для экономии ввода можно поступить и иначе, используя инициализацию структуры при ее объявлении.

```
WNDCLASSEX winclass = {
    sizeof(WNDCLASSEX),
    CS_VREDRAW | CS_HREDRAW | CS_OWNDC | CS_DBLCLKICKS,
    WinProc,
    0,
    0,
    hInstance,
    LoadIcon(NULL, IDI_APPLICATION),
    LoadCursor(NULL, IDC_ARROW),
    (HBRUSH)GetStockObject(WHITE_BRUSH),
    NULL,
    "WINCLASS1",
    LoadIcon(NULL, IDI_APPLICATION)};
```

## Регистрация класса Windows

Теперь, когда класс Windows определен и сохранен в переменной `winClass`, его следует зарегистрировать. Для этого вызывается функция `RegisterClassEx()`, которой передается указатель на определение нового класса.

```
RegisterClassEx(&winclass);
```

Я думаю, очевидно, что здесь мы используем не имя класса, а структуру, описывающую его свойства, — ведь до этого вызова Windows не подозревает о существовании нового класса вообще.

Для полноты изложения упомяну также устаревшую функцию `RegisterClass()`, которой передается указатель на структуру `WNDCLASS`.

После того как класс зарегистрирован, можно создавать соответствующие окна. Выясним, как это делается, а затем детально рассмотрим обработчик событий, главный цикл событий и узнаем, какого рода обработку событий следует обеспечить для работоспособности приложения Windows.

## Создание окна

Для создания окна (или любого другого “окнообразного” объекта) используется функция `CreateWindow()` или `CreateWindowEx()`. Последняя функция более новая и поддерживает дополнительные параметры стиля, так что будем использовать именно ее. При создании окна требуется указать текстовое имя класса данного окна (в нашем случае это “`WNDCLASS1`”).

Вот как выглядит прототип функции `CreateWindowEx()`.

```
HWND CreateWindowEx(
    DWORD dwExStyle, // Дополнительный стиль окна
    LPCTSTR lpClassName, // Указатель на имя
                        // зарегистрированного класса
    LPCTSTR lpWindowName, // Указатель на имя окна
    DWORD dwStyle, // Стиль окна
    int x, // Горизонтальная позиция окна
    int y, // Вертикальная позиция окна
    int nWidth, // Ширина окна
    int nHeight, // Высота окна
    HWND hWndParent, // Дескриптор родительского окна
    HMENU hMenu, // Дескриптор меню или
                // идентификатор дочернего окна
    HINSTANCE hInstance, // Дескриптор экземпляра приложения
    LPVOID lpParam); // Указатель на данные создания окна
```

Если функция выполнена успешно, она возвращает дескриптор вновь созданного окна; в противном случае возвращается значение `NULL`.

Большинство параметров очевидны; тем не менее вкратце рассмотрим все параметры функции.

- `dwExStyle`. Флаг дополнительных стилей окна; в большинстве случаев просто равен `NULL`. Если вас интересует, какие именно значения может принимать данный параметр, обратитесь к справочной системе Win32 SDK. Единственный из этих параметров, время от времени использующийся мною, — `WS_EX_TOPMOST`, который “заставляет” окно находиться поверх других.
- `lpClassName`. Имя класса, на основе которого создается окно.
- `lpWindowName`. Завершающаяся нулевым символом строка с заголовком окна, например “Мое первое окно”.
- `dwStyle`. Флаги, описывающие, как должно выглядеть и вести себя создаваемое окно. Этот параметр очень важен! В табл. 2.7 приведены некоторые наиболее часто используемые флаги (которые, как обычно, можно объединять с помощью операции побитового ИЛИ).

- **x, y.** Позиция верхнего левого угла окна в пикселях. Если положение окна при создании не принципиально, воспользуйтесь значением **CW\_USEDEFAULT**, и Windows разместит окно самостоятельно.
- **nWidth, nHeight.** Ширина и высота окна в пикселях. Если размер окна при создании не принципиален, воспользуйтесь значением **CW\_USEDEFAULT**, и Windows выберет размеры окна самостоятельно.
- **hWndParent.** Дескриптор родительского окна, если таковое имеется. Значение **NULL** указывает, что у создаваемого окна нет родительского окна и в качестве такового должна считаться поверхность рабочего стола.
- **hMenu.** Дескриптор меню, присоединенного к окну. О нем речь идет в следующей главе, а пока будем использовать значение **NULL**.
- **hInstance.** Экземпляр приложения. Здесь используется значение параметра **hInstance** функции **WinMain()**.
- **lpParam.** Пока что мы просто устанавливаем это значение равным **NULL**.

**Таблица 2.7. Значения стилей, использующиеся в параметре *dwStyle***

<i>Значение</i>	<i>Описание</i>
<b>WS_POPUP</b>	Всплывающее окно
<b>WS_OVERLAPPED</b>	Перекрывающееся окно с полосой заголовка и рамкой. То же, что и стиль <b>WS_TILED</b>
<b>WS_OVERLAPPEDWINDOW</b>	Перекрывающееся окно со стилями <b>WS_OVERLAPPED</b> , <b>WS_CAPTION</b> , <b>WS_SYSMENU</b> , <b>WS_THICKFRAME</b> , <b>WS_MINIMIZEBOX</b> и <b>WS_MAXIMIZEBOX</b>
<b>WS_VISIBLE</b>	Изначально видимое окно
<b>WS_SYSMENU</b>	Окно с меню в полосе заголовка. Требует также установки стиля <b>WS_CAPTION</b>
<b>WS_BORDER</b>	Окно в тонкой рамке
<b>WS_CAPTION</b>	Окно с полосой заголовка (включает стиль <b>WS_BORDER</b> )
<b>WS_ICONIC</b>	Изначально минимизированное окно. То же, что и стиль <b>WS_MINIMIZE</b>
<b>WS_MAXIMIZE</b>	Изначально максимизированное окно
<b>WS_MAXIMIZEBOX</b>	Окно с кнопкой Maximize. Не может быть скомбинирован со стилем <b>WS_EX_CONTEXTHELP</b> ; кроме того, требуется указание стиля <b>WS_SYSMENU</b>
<b>WS_MINIMIZE</b>	Изначально минимизированное окно. То же, что и стиль <b>WS_ICONIC</b>
<b>WS_MINIMIZEBOX</b>	Окно с кнопкой Minimize. Не может быть скомбинирован со стилем <b>WS_EX_CONTEXTHELP</b> ; кроме того, требуется указание стиля <b>WS_SYSMENU</b>
<b>WS_POPUPWINDOW</b>	Всплывающее окно со стилями <b>WS_BORDER</b> , <b>WS_POPUP</b> и <b>WS_SYSMENU</b> . Для того чтобы меню окна было видно, требуется комбинация стилей <b>WS_CAPTION</b> и <b>WS_POPUPWINDOW</b>
<b>WS_SIZEBOX</b>	Окно, размер которого можно изменять перетягиванием рамки. То же, что и <b>WS_THICKFRAME</b>
<b>WS_HSCROLL</b>	Окно имеет горизонтальную полосу прокрутки
<b>WS_VSCROLL</b>	Окно имеет вертикальную полосу прокрутки

*Примечание.* Наиболее часто употребляющиеся значения выделены полужирным шрифтом.

Вот как создается обычное **перекрывающееся** окно со стандартными управляющими элементами в **позиции** (0,0) размером 400х400 пикселей.

```
HWND hwnd; // Дескриптор окна
if (!hwnd - CreateWindowEx( NULL,
    "WINCLASS1",
    "Your Basic Window",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0, 0, 400, 400,
    NULL, NULL,
    hinstance, NULL)))
return (0);
```

После того как окно создано, оно может быть как видимым, так и невидимым. В нашем случае мы добавили флаг **WS\_VISIBLE**, который делает окно видимым автоматически. Если этот флаг не добавлен, требуется явным образом вывести окно на экран.

```
// Вывод окна на экран
ShowWindow(hwnd, ncmdshow);
```

Помните параметр **ncmdshow** функции **WinMain()**? Вот где он пригодился. Хотя в нашем случае просто использовался флаг **WS\_VISIBLE**, обычно этот параметр передается функции **ShowWindow()**. Следующее, что вы можете захотеть сделать, — это заставить Windows обновить содержимое окна и сгенерировать сообщение **WM\_PAINT**. Все это делается вызовом функции **UpdateWindow()**, которая не получает никаких параметров.

```
// Этот код посылает сообщение WM_PAINT окну
//и обеспечивает обновление его содержимого
UpdateWindow();
```

## Обработчик событий

Для того чтобы освежить память и вспомнить о том, для чего нужен обработчик событий и что такое функция обратного вызова, вернитесь к рис. 2.3.

Обработчик событий создается вами и обрабатывает столько событий, сколько вы сочтете необходимым. Всеми остальными событиями, остающимися необработанными, будет заниматься Windows. Разумеется, чем больше событий и сообщений обрабатывает ваша программа, тем выше ее функциональность.

Прежде чем приступить к написанию кода, обсудим детали обработчика событий и выясним, что он собой представляет и как работает. Для каждого создаваемого класса Windows вы можете определить свой собственный обработчик событий, на который в дальнейшем я буду ссылаться как на *процедуру Windows* или просто *WinProc*. В процессе работы пользователя (и самой операционной системы Windows) для вашего окна, как и для окон других приложений, генерируется масса событий и **сообщений**. Все эти сообщения попадают в очередь, причем сообщения для вашего окна попадают в очередь сообщений вашего окна. Главный цикл обработки сообщений изымает их из очереди и **передает** WinProc вашего окна для обработки.

Существуют сотни возможных сообщений, так что обработать их все мы просто не **остановимся**. К счастью, для того, чтобы получить работоспособное приложение, мы можем обойтись только небольшой их частью.

Итак, главный цикл обработки событий передает сообщения и события WinProc, которая **выполняет** с ними некоторые действия. Следовательно, мы должны побеспокоиться только о WinProc, но и о главном цикле обработки событий.

Рассмотрим теперь прототип WinProc.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd, // Дескриптор окна или отправитель
    UINT msg,   // Идентификатор сообщения
    WPARAM wParam, // Дополнительная информация о сообщении
    LPARAM lParam); // Дополнительная информация о сообщении

```

Разумеется, это просто прототип функции обратного вызова, которую вы можете называть как угодно, лишь бы ее адрес был присвоен полю winclass.lpfnWndProc.

```
winclass.lpfnWndProc = WindowProc;
```

А теперь познакомимся с параметрами этой функции.

- **hwnd**. Дескриптор окна. Этот параметр важен в том случае, когда открыто несколько окон одного и того же класса. Тогда **hwnd** позволяет определить, от какого именно окна поступило сообщение (рис. 2.4).
- **msg**. Идентификатор сообщения, которое должно быть обработано WinProc.
- **wparam** и **lparam**. Эти величины являются параметрами обрабатываемого сообщения, несущими дополнительную информацию о нем.

И никогда не забывайте о спецификации CALLBACK при объявлении данной функции!

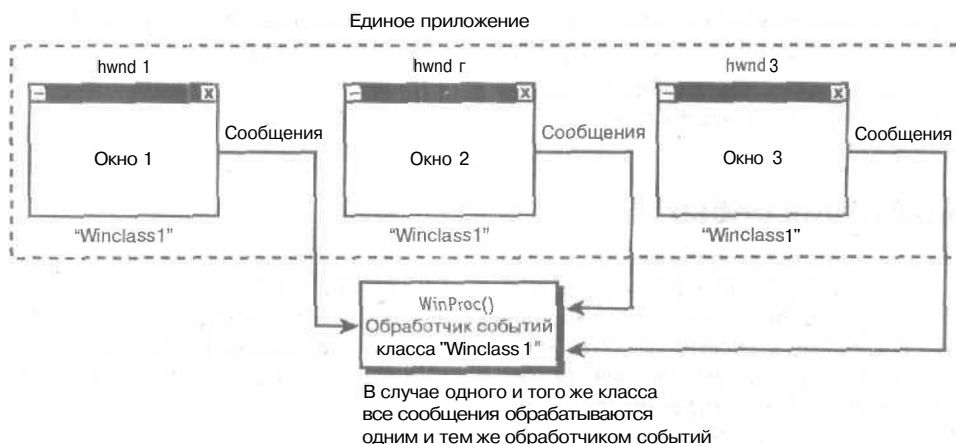


Рис. 2.4. Несколько окон одного и того же класса

Большинство программистов при написании обработчика событий используют конструкцию `switch(msg)`, в которой на основании значения `msg` принимается решение об использовании дополнительных параметров `wparam` и/или `lparam`. В табл. 2,8 приведены некоторые из возможных идентификаторов сообщений.

Таблица 2.8. Краткий список идентификаторов сообщений

Значение	Описание
WM_ACTIVATE	Посылается, когда окно активизируется или получает фокус ввода
WM_CLOSE	Посылается, когда окно закрывается
WM_CREATE	Посылается при создании окна

Значение	Описание
WM_DESTROY	Посылается, когда окно должно быть уничтожено
WM_MOVE	Посылается при перемещении окна
WM_MOUSEMOVE	Посылается при перемещении мыши
WM_KEYUP	Посылается при отпускании клавиши
WM_KEYDOWN	Посылается при нажатии клавиши
WM_TIMER	Посылается при наступлении события таймера
WM_USER	Позволяет вам посылать собственные сообщения
WM_PAINT	Посылается при необходимости перерисовки окна
WM_QUIT	Посылается при завершении работы приложения
WM_SIZE	Посылается при изменении размеров окна

Внимательно посмотрите на приведенные в таблице идентификаторы сообщений. При работе вам придется иметь дело в основном с этими сообщениями. Идентификатор сообщения передается в WinProc в виде параметра `msg`, а вся сопутствующая информация — как параметры `wparam` и `lparam`. Какая именно информация передается в этих параметрах для каждого конкретного сообщения, можно узнать из справочной системы Win32 SDK.

При разработке игр нас в первую очередь интересуют три типа сообщений.

- **WM\_CREATE.** Это сообщение посылается при создании окна и позволяет нам выполнить все необходимые действия по инициализации, захвату ресурсов и т.п.
- **WM\_PAINT.** Это сообщение посылается, когда требуется перерисовка окна. Это может произойти по целому ряду причин: окно было перемещено, его размер был изменен, окно другого приложения перекрыло наше окно и т.п.
- **WM\_DESTROY.** Это сообщение посылается нашему окну перед тем, как оно должно быть уничтожено. Обычно это сообщение — результат щелчка на пиктограмме закрытия окна или выбор соответствующего пункта системного меню. В любом случае по получении этого сообщения следует освободить все захваченные ресурсы и сообщить Windows о необходимости закрыть приложение, послав сообщение **WM\_QUIT** (мы встретимся с этим немного позже в данной главе).

Вот простейшая функция WinProc, обрабатывающая описанные сообщения.

```

LRESULT CALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wparam,
    LPARAM lparam)
{
    PAINTSTRUCT ps; // Используется в WM_PAINT
    HDC hdc; // Дескриптор контекста устройства

    // Какое сообщение получено?
    switch(msg)
    {
        case WM_CREATE:
            { // Выполнение инициализации

```

```

        return(0); // Успешное выполнение
    } break;

    case WM_PAINT:
    {
        // Обновляем окно
        hdc = BeginPaint(hwnd,&ps);
        // Здесь выполняется перерисовка окна
        EndPaint(hwnd,&ps);

        return(0); // Успешное выполнение
    } break;

    case WM_DESTROY:
    {
        // Завершение приложения
        PostQuitMessage(0);
        return(0); // Успешное выполнение
    } break;

    default: break;

} // switch

// Обработка прочих сообщений
return(DefWindowProc(hwnd, msg, wparam, tparam));

} // WindowProc

```

Как видите, обработка сообщений в основном сводится к отсутствию таковой :-). Начнем с сообщения `WM_CREATE`. Здесь наша функция просто возвращает нулевое значение, говорящее Windows, что сообщение успешно обработано и никаких других действий предпринимать не надо. Конечно, здесь можно выполнить все действия по инициализации, однако в настоящее время у нас нет такой необходимости.

Сообщение `WM_PAINT` очень важное. Оно посылается при необходимости перерисовки окна. В случае игр DirectX это не так важно, поскольку мы перерисовываем весь экран со скоростью от 30 до 60 раз в секунду, но для обычных приложений Windows это может иметь большое значение. Более детально `WM_PAINT` рассматривается в следующей главе, а пока просто сообщим Windows, что мы уже перерисовали окно, так что посылать сообщение `WM_PAINT` больше не требуется.

Для этого необходимо объявить действительной клиентскую область окна. Возможно несколько путей решения этой задачи, и простейший из них — использовать пару вызовов `BeginPaint()` — `EndPaint()`. Эта пара вызовов делает окно действительным и заливает фон с помощью выбранной в определении класса кисти.

```

        hdc = BeginPaint(hwnd,&ps);
        // Здесь выполняется перерисовка окна
        EndPaint(hwnd,&ps);

```

Здесь я хотел бы кое-что подчеркнуть. Заметьте, что первым параметром в каждом вызове является дескриптор окна `hwnd`. Это необходимо, поскольку функции `BeginPaint()` и `EndPaint()` потенциально способны выводить изображение в любом окне вашего приложения и дескриптор указывает, в какое именно окно будет перерисовываться. Второй параметр представляет собой указатель на структуру `PAINTSTRUCT`, содержащую прямоугольник, который необходимо перерисовать. Эта структура представлена ниже.

```
typedef struct tagPAINTSTRUCT
{
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```



Рис. 2.5. Перерисовывается только недействительная область

Сейчас, пока не рассматривалась работа с GDI, вам незачем беспокоиться о всех полях этой структуры — вы познакомитесь с ними позже. Здесь же стоит познакомиться поближе только с полем `rcPaint`, которое представляет собой структуру `RECT`, указывающую минимальный прямоугольник, требующий перерисовки. Взгляните на рис. 2.5, поясняющий это. Windows пытается обойтись минимальным количеством действий, поэтому при необходимости перерисовки окна старается вычислить минимальный прямоугольник, перерисовать который было бы достаточно для восстановления содержимого всего окна. Если вас интересует, что конкретно представляет собой структура `RECT`, то это не более чем четыре угла прямоугольника.

```
typedef struct tagRECT
{
    LONG left; // Левая сторона прямоугольника
    LONG top; // Верхняя сторона прямоугольника
    LONG right; // Правая сторона прямоугольника
    LONG bottom; // Нижняя сторона прямоугольника
} RECT;
```

Последнее, что следует сказать о вызове `BeginPaint()`: эта функция возвращает дескриптор графического контекста `hdc`.

```
HDC hdc;
hdc = BeginPaint(hwnd, &ps);
```

*Графический контекст* представляет собой структуру данных, которая описывает видеосистему и изображаемую поверхность. Для *непосвященного* — это сплошное шаманство, так что запомните главное: вы должны получить этот контекст, если хотите работать с какой-либо графикой. На этом мы пока завершаем рассмотрение сообщения WM\_PAINT.

Сообщение WM\_DESTROY достаточно интересно. Оно посылается, когда пользователь закрывает окно. Однако это действие закрывает только окно, но не завершает само приложение, которое продолжает работать, но уже без окна. В большинстве случаев, когда пользователь закрывает окно, он намерен завершить работу приложения, так что вам надо заняться этим и послать сообщение о завершении приложения самому себе. Это сообщение WM\_QUIT, и в связи с распространенностью его использования имеется даже специальная функция для отправки его самому себе — PostQuitMessage().

При обработке сообщения WM\_DESTROY необходимо выполнить все действия по освобождению ресурсов и сообщить Windows, что она может завершать работу вашего приложения, вызвав PostQuitMessage(0). Этот вызов поместит в очередь сообщение WM\_QUIT, которое приведет к завершению работы главного цикла событий.

При анализе WinProc вы должны знать некоторые детали. Во-первых, я уверен, что вы уже обратили внимание на операторы return(0) после обработки каждого сообщения. Этот оператор служит для двух целей: выйти из WinProc и сообщить Windows, что вы обработали сообщение. Вторая важная деталь состоит в использовании *обработчика сообщений по умолчанию* DefaultWindowProc(). Эта функция передает необработанные вами сообщения Windows для обработки по умолчанию. Таким образом, если вы не обрабатываете какие-то сообщения, то всегда завершайте ваш обработчик событий вызовом

```
return(DefaultWindowProc(hwnd, msg, wparam, lparam));
```

Я знаю, что все это может показаться сложным. Тем не менее, все очень просто: достаточно иметь основу кода приложения Windows и просто добавлять к ней собственный код. Моя главная цель, как я уже говорил, состоит в том, чтобы помочь вам в создании “DOS32-подобных” игр, где вы можете практически забыть о существовании Windows. В любом случае, нам необходимо рассмотреть еще одну важную часть Windows-приложения — главный цикл событий.

## Главный цикл событий

Все, сложности закончились, так как главный цикл событий очень прост.

```
while(GetMessage(&msg, NULL, 0, 0))
{
    // Преобразование клавиатурного ввода
    TranslateMessage(&msg);

    // Пересылка сообщения WinProc
    DispatchMessage(&msg);
}
```

Вот и все. Цикл while() выполняется до тех пор, пока GetMessage() возвращает ненулевое значение. Эта функция и есть главная рабочая лошадь цикла, единственная цель которого состоит в извлечении очередного сообщения из очереди событий и его обработке. Обратите внимание на то, что функция GetMessage() имеет четыре параметра. Для нас важен только первый параметр; остальные имеют нулевые значения. Ниже представлен прототип функции GetMessage().

```
BOOL GetMessage(
    LPMSG lpMsg,    // Адрес структуры с сообщением
```

```

HWND hWnd,    // Дескриптор окна
UINT wParam,  // Первое сообщение
UINT lParam;  // Последнее сообщение

```

Параметр `msg` представляет собой переменную для хранения полученного сообщения. Однако, в отличие от параметра `msg` в `WinProc()`, этот параметр представляет собой сложную структуру данных.

```
typedef struct tagMSG
```

```

{
    HWND hwnd; // Окно сообщения
    UINT message; // Идентификатор сообщения
    WPARAM wParam; // Дополнительный параметр сообщения
    LPARAM lParam; // Дополнительный параметр сообщения
    DWORD time; // Время события сообщения
    POINT pt; // Положение указателя мыши
} MSG;

```

Итак, как видите, все параметры функции `WinProc()` содержатся в этой структуре данных наряду с другой информацией, такой как время или положение указателя мыши.

Итак, функция `GetMessage()` получает очередное сообщение из очереди событий. А что затем? Следующей вызывается функция `TranslateMessage()`, которая представляет собой транслятор виртуальных “быстрых клавиш”. Ваше дело — **вызвать** эту функцию, остальное — не ваша забота. Последняя вызываемая функция — `DispatchMessage()`, которая, собственно, и выполняет всю работу по обработке сообщений. После того как сообщение получено с помощью функции `GetMessage()` и преобразовано функцией `TranslateMessage()`, функция `DispatchMessage()` вызывает для обработки функцию `WinProc()`, передавая ей всю необходимую информацию, которая находится в структуре `MSG`. На рис. 2.6 показан весь описанный процесс.

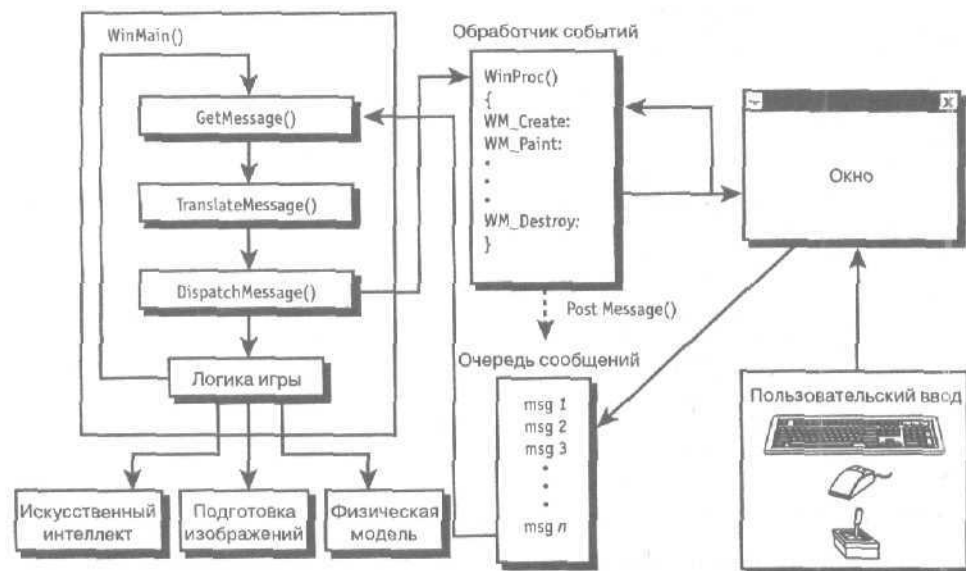


Рис. 2.6. Механизм обработки сообщений

Вот и все; если вы разобрались в этом, можете считать себя программистом в Windows. Остальное — не более чем небольшие детали. Посмотрите с этой точки зрения

на приведенный ниже листинг, где представлена завершенная Windows-программа, которая просто создает одно окно и ждет, пока вы его не закроете.

// DEM02\_3.CPP - Базовая Windows-программа

```
// Включаемые файлы //////////////////////////////////////
#define WIN32_LEAN_AND_MEAN // Не MFC :-)
```

```
tfincLude <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include <math.h>
```

```
// Определения //////////////////////////////////////
```

```
#define WINDOW_CLASS_NAME "WINCLASS1"
```

```
// Глобальные переменные //////////////////////////////////////
```

```
// Функции //////////////////////////////////////
```

```
LRESULT CALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wparam,
    LPARAM tparam)
```

```
{
    // Главный обработчик сообщений в системе
    PAINTSTRUCT ps; // Используется в WM_PAINT
    HDC hdc; // Дескриптор контекста устройства
    // Какое сообщение получено
    switch(msg)
```

```
{
    case WM_CREATE:
```

```
    {
        // Действия по инициализации
        return(0); // Успешное завершение
    } break;
```

```
    case WM_PAINT:
```

```
    {
        // Объявляем окно действительным
        hdc = BeginPaint(hwnd,&ps);
        // Здесь выполняются все действия
        // по выводу изображения
        EndPaint(hwnd,&ps);
        return(0); // Успешное завершение
    } break;
```

```
    case WM_DESTROY:
```

```
    {
        // Завершение работы приложения
        PostQuitMessage(0);
```

```
        // Успешное завершение
        return(0);
    }
}
```

```

        } break;
        default: break;
    } // switch
    // Обработка остальных сообщений
    return (DefWindowProc(hwnd, msg, wParam, lParam));
} // WinProc

//WinMain/////////////////////////////////////////////////////////////////
int WINAPI WinMain( HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
                   LPSTR lpcmdline,
                   int ncmdshow)
{
    WNDCLASSEX winclass; //Класс создаваемого сообщения
    HWND hwnd; // Дескриптор окна
    MSG msg; // Структура сообщения

    // Заполнение структуры класса
    winclass.cbSize = sizeof(WNDCLASSEX);
    winclass.style = CS_DBLCLKS | CS_OWNDC |
        CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = hinstance;
    winclass.hIcon =
        LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground =
        (HBRUSH)GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;
    winclass.hIconSm =
        LoadIcon(NULL, IDI_APPLICATION);

    // Регистрация класса
    if (!RegisterClassEx(&winclass))
        return(0);
    // Создание окна
    if (!(hwnd =
        CreateWindowEx(NULL,
            WINDOW_CLASS_NAME,
            "Your Basic Window",
            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
            0,
            400, 400,
            NULL,
            NULL,
            hinstance,
            NULL)))
        return(0);
}

```

```
// Главный цикл событий
while(GetMessage(&msg,NULL,0,0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
} // while

// Возврат в Windows
return(msg.wParam);
} // WinMain
```

////////////////////////////////////

Для компиляции программы создайте проект Win32 .EXE и добавьте к нему DEMO2\_3.CPP (вы можете также просто запустить скомпилированную программу DEMO2\_3.EXE, находящуюся на прилагаемом компакт-диске). На рис. 2.7 показана работа данной программы.

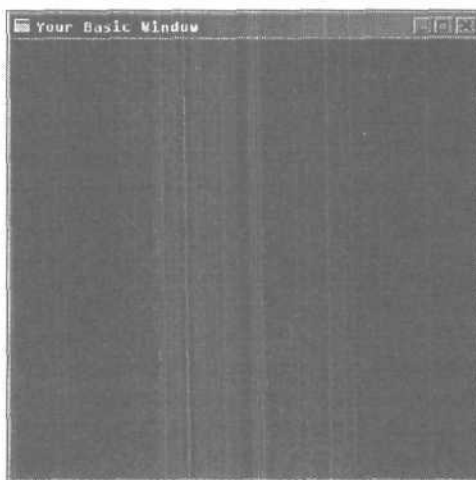


Рис. 2.7. Копия экрана приложения DEMO2\_3.EXE

Прежде чем идти дальше, хотелось бы осветить ряд вопросов. Начнем с того, что если вы внимательнее посмотрите на цикл событий, то увидите, что он не предназначен для работы в реальном времени; ведь пока программа ожидает получения сообщения посредством функции GetMessage(), ее выполнение заблокировано.

## Создание цикла событий реального времени

Для решения этой проблемы нам нужен способ узнать, **имеется** ли какое-либо сообщение в очереди или она пуста. Если сообщение есть, мы обрабатываем его; если нет — продолжаем выполнение игры. Windows предоставляет необходимую нам функцию PeekMessage(). Ее прототип практически идентичен прототипу функции GetMessage().

```
BOOL PeekMessage(
    LPMSG lpMsg,    // Адрес структуры с сообщением
    HWND hWnd,     // Дескриптор окна
    UINT wMsgFilterMin, // Первое сообщение
```

```
UINT wMsgFilterMax, // Последнее сообщение
UINT wRemoveMsg); // Флаг удаления сообщения
```

Данная функция **возвращает** ненулевое значение, если в очереди имеется сообщение.

Отличие прототипов функций заключается в последнем **парамetre**, который определяет, как именно должно быть выбрано сообщение из очереди. Корректными значениями параметра `wRemoveMsg` являются следующие:

- `PM_NOREMOVE`. Вызов `PeekMessage()` не удаляет сообщение из очереди,
- `PM_REMOVE`. Вызов `PeekMessageQ` удаляет сообщение из очереди.

Рассматривая варианты вызова функции `PeekMessage()`, мы приходим к двум возможным способам работы: либо вызываем функцию `PeekMessage()` с параметром `PM_NOREMOVE` и, если в очереди имеется сообщение, вызываем функцию `GetMessage()`; либо используем функцию `PeekMessage()` с параметром `PM_REMOVE`, чтобы сразу выбрать сообщение из очереди, если оно там есть. Воспользуемся именно этим способом. Вот измененный основной цикл, отражающий принятую нами методику работы.

```
while(TRUE)
{
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        // Проверка сообщения о выходе
        if (msg.message == WM_QUIT) break;

        TranslateMessage(&msg);

        DispatchMessage(&msg);
    } // if

    // Выполнение игры
    Game_Main();
} //while
```

В приведенном коде наиболее важные моменты выделены полужирным **шрифтом**. Рассмотрим первую выделенную часть.

```
if (msg.message == WM_QUIT) break;
```

Этот код позволяет определить, когда следует выйти из бесконечного цикла `while(TRUE)`. Вспомните: когда в `WinProc` обрабатывается сообщение `WM_DESTROY`, мы посылаем сами себе сообщение `WM_QUIT` посредством вызова функции `PostQuitMessage()`. Это сообщение знаменует завершение работы программы, и, получив его, мы **выходим** из основного цикла.

Вторая выделенная полужирным шрифтом часть кода указывает, где следует разместить вызов главного цикла игры. Но не забывайте о том, что возврат из вызова `Game_Main()` — или как вы его там назовете — должен осуществляться немедленно по выводу очередного кадра анимации или одного просчета логики игры. В противном случае обработка сообщений в главном цикле прекратится.

Примером использования нового подхода к решению проблемы цикла реального времени может служить программа `DEMO2_4.CPP`, которую вы можете найти на прилагаемом компакт-диске. Именно эта структура программы и будет использоваться в оставшейся части книги.

## Краткий курс DirectX и COM

DirectX может потребовать большего количества работы и вмешательства со стороны программиста, но дело стоит того. DirectX представляет собой программное обеспечение, которое позволяет абстрагировать видеоинформацию, звук, входящую информацию, работу в сети и многое другое таким образом, что аппаратная конфигурация компьютера перестает иметь значение и для любого аппаратного обеспечения используется один и тот же программный код. Кроме того, технология DirectX более высокоскоростная и надежная, чем GDI или MCI (Media Control Interface — интерфейс управления средой), являющиеся "родными" технологиями Windows,

На рис. 2.8 показаны схемы разработки игры для Windows с использованием DirectX и без нее. Обратите внимание, насколько ясным и элегантным решением является DirectX.

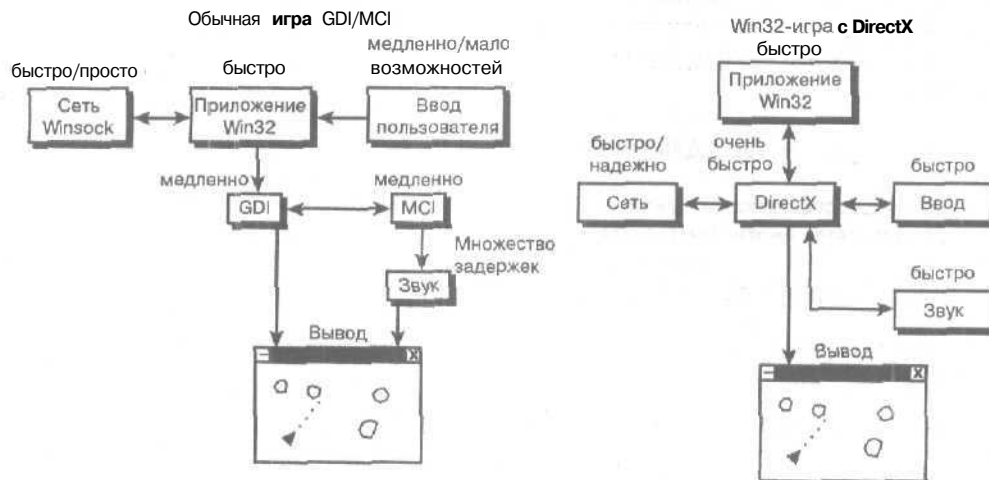


Рис. 2.8. DirectX против GDI/MCI

Так как же работает DirectX? Эта технология предоставляет возможность управления всеми устройствами практически на аппаратном уровне. Это возможно благодаря технологии COM (Component Object Model — модель составных объектов) и множеству драйверов и библиотек, написанных как Microsoft, так и производителями оборудования. Microsoft продумала и сформулировала ряд соглашений: функции, переменные, структуры данных и т.д. — словом, все то, чем должны пользоваться производители оборудования при разработке драйвера для управления производимым ими устройством.

Если эти соглашения соблюдаются, вам не следует беспокоиться об особенностях того или иного устройства. Достаточно просто обратиться к DirectX, а уж она сама обработает и учтет эти особенности за вас. Если у вас есть поддержка DirectX, то совершенно не важно, какая у вас звуковая или видеокарта либо другие устройства. Любая программа может обращаться к тому или иному устройству, даже не имея никакого представления о нем.

В настоящий момент DirectX состоит из ряда компонентов. Их список приводится ниже и показан на рис. 2.9.

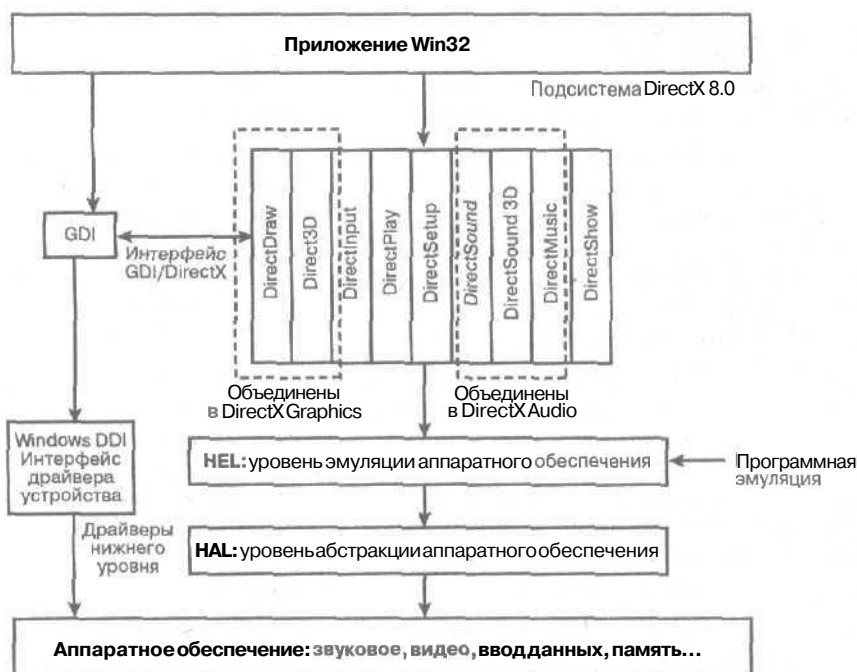


Рис. 2.9. Архитектура DirectX и отношения с Win32

- DirectDraw
- DirectSound
- DirectSound3D
- DirectMusic
- DirectInput
- DirectPlay
- DirectSetup
- Direct3DRM
- Direct3DIM

В DirectX 8.0 DirectDraw был слит с Direct3D и переименован в DirectGraphics, а DirectSound был слит с DirectMusic и переименован в DirectAudio. Таким образом, в DirectX 8.0+ больше нет DirectDraw. Однако одно из правил DirectX и COM состоит в том, что вы всегда можете запросить предыдущий интерфейс. Таким образом, в этой книге мы будем использовать для графики DirectDraw под DirectX 7.0a, а для звука — новейшие интерфейсы 8.0+. В любом случае мы собираемся собрать графику, звук и средства ввода в нашем виртуальном компьютере, так что будет работать даже DirectX 3.0.

## HEL и HAL

На рис. 2.9 можно заметить, что DirectX основывается на двух уровнях, которые называются HEL (Hardware Emulation Layer— уровень эмуляции аппаратного обеспечения) и HAL (Hardware Abstraction Layer— уровень абстракции аппаратного обеспечения).

Так как DirectX создан "с дальним прицелом", предполагается, что в будущем аппаратное обеспечение будет поддерживать дополнительные возможностями, которыми можно будет пользоваться при работе с DirectX. Ну, а что делать, пока аппаратное обеспечение не поддерживает эти возможности? Проблемы этого рода и призваны решать HEL и HAL.

- \* HAL находится ближе к "железу" и общается с устройствами напрямую. Обычно
  - это драйвер устройства, написанный производителем, и с помощью обычных запросов DirectX вы обращаетесь непосредственно к нему. Отсюда вывод: HAL используется тогда, когда вам надо обратиться к функции, поддерживаемой самим устройством (что существенно ускоряет работу). Например, если вы хотите загрузить растровый рисунок на экран, то аппаратно это будет сделано гораздо быстрее, чем при использовании программного цикла.
  - HEL используется тогда, когда устройство не поддерживает необходимую вам функцию. Например, вы хотите заставить изображение на экране вращаться. Если видеоадаптер не поддерживает вращение на аппаратном уровне, на арену выходит HEL и выполнение этой функции берет на себя программное обеспечение. Понятно, что этот способ работает медленнее, но худо-бедно, а ваша программа продолжает работать. При этом вы даже не знаете, на каком именно уровне выполнено ваше задание. Если задача решается на уровне HAL, это будет сделано на аппаратном уровне. В противном случае для выполнения вашего задания будет вызвана программа из HEL.

Вы можете решить, что в HEL очень много уровней программного обеспечения. Это так, но это не должно вас волновать: DirectX — настолько ясная технология, что единственное неудобство для программиста заключается лишь в вызове одной-двух лишних функций. Это не слишком большая плата за ускорение 2D/3D графики, работы в сети и обработки звука. DirectX — это попытка Microsoft и производителей оборудования помочь вам полностью использовать все аппаратные возможности.

## Подробнее о базовых классах DirectX

Теперь вкратце познакомимся с компонентами DirectX и узнаем, чем занимается каждый из них.

**DirectDraw.** Основной компонент, отвечающий за вывод двумерных изображений и управляющий дисплеем. Это тот канал, по которому проходит вся графика и который, пожалуй, является самым важным среди всех компонентов DirectX. Объект DirectDraw в большей или меньшей степени представляет видеокарту вашей системы. Однако в DirectX 8.0 он более не доступен, так что для этой цели следует пользоваться интерфейсами DirectX 7.0.

**DirectSound.** Компонент DirectX, отвечающий за звук. Он поддерживает только цифровой звук, но не MIDI. Однако использование этого компонента значительно упрощает жизнь, так как теперь вам не нужна лицензия на использование звуковых систем сторонних производителей. Программирование звука — это настоящая черная магия, и на рынке звуковых библиотек ряд производителей загнали в угол всех остальных, выпустив Miles Sound System и DiamondWare Sound Toolkit. Это были очень удачные системы, позволявшие легко загружать и проигрывать и цифровой звук, и MIDI как под DOS, так и из Win32-программ. Однако благодаря DirectSound, DirectSound3D и новейшему компоненту DirectMusic библиотеки сторонних разработчиков используются все реже.

**DirectSound3D.** Компонент DirectSound, отвечающий за 3D-звук. Он позволяет позиционировать звук в пространстве таким образом, чтобы создавалось впечатление движения объектов. Это достаточно новая, но быстро совершенствующаяся технология.

На сегодня звуковые платы поддерживают 3D-эффекты на аппаратном уровне, включая такие эффекты, как эффект Доплера, преломление, отражение и др. Однако при использовании программной эмуляции от всех этих возможностей остается едва ли половина.

**DirectMusic.** Самое последнее новшество в DirectX. Поддерживает технологию MIDI, незаслуженно забытую ранее. Кроме того, в DirectX есть новая система под названием DLS (Downloadable Sounds System — система подгружаемых звуков), которая позволяет создавать цифровое представление музыкальных инструментов, а затем использовать его в MIDI-контроллере. Это во многом подобно синтезатору Wave Table, но только работающему на программном уровне. DirectMusic позволяет также в реальном времени изменять параметры звука, пользуясь вашими шаблонами. По существу, эта система в состоянии создавать новую музыку "на лету".

**DirectInput.** Система, которая обрабатывает информацию, поступающую со всех устройств ввода, включая мышь, клавиатуру, джойстик, пульт ручного управления, манипулятор-шар и т.д. Кроме того, в настоящее время DirectInput поддерживает электромеханические приводы и датчики, определяющие силу давления, что дает возможность задавать механическую силу, которую пользователь ощущает физически. Эти возможности могут буквально взорвать индустрию киберсекса! :-)

**DirectPlay.** Часть DirectX, работающая с сетью. Использование DirectPlay позволяет абстрагировать сетевые подключения, использующие Internet, модемы, непосредственное соединение компьютер или любые другие типы соединений, которые могут когда-либо появиться. DirectPlay позволяет работать с подключениями любого типа, даже не имея ни малейшего представления о работе в сети. Вам больше не придется писать драйверы, использовать сокет или что-либо в этом роде. Кроме того, DirectPlay поддерживает концепции сессии, т.е. самого процесса игры, и того места в сети, где игроки собираются для игры. DirectPlay не требует от вас знания многопользовательской архитектуры. Отправка и получение пакетов для вас — вот все, что он делает. Содержимое и достоверность этих пакетов — ваше дело.

**Direct3DRM (DirectX3D Retained Mode** — режим поддержки Direct3D). Высокоуровневая 3D-система, основанная на объектах и фреймах, которую можно использовать для создания базовых 3D-программ. Direct3DRM использует все достоинства 3D-ускорителей, хотя и не является самой быстрой системой трехмерной графики в мире.

**Direct3DIM (Direct3D Immediate Mode** — режим непосредственной поддержки Direct3D). Представляет собой низкоуровневую поддержку трехмерной графики DirectX. Первоначально с ним было невероятно трудно работать и это было слабым местом в войне DirectX с OpenGL. Старая версия Immediate Mode использовала так называемые *execute buffers* (буферы выполнения). Эти созданные вами массивы данных и команд, описывающие сцену, которая должна быть нарисована, — не самая красивая идея. Однако, начиная с DirectX 5.0, интерфейс Immediate Mode больше напоминает интерфейс OpenGL благодаря функции `DrawPrimitive()`. Теперь вы можете передавать обработчику отдельные детали изображения и изменять состояния при помощи вызовов функций, а не посредством буферов выполнения. Честно говоря, я полюбил Direct3D Immediate Mode только после появления в нем указанных возможностей. Однако в данной книге мы не будем углубляться в детали использования Direct3DIM.

**DirectSetup/AutoPlay.** Квазикомпоненты DirectX, обеспечивающие установку DirectX на машину пользователя непосредственно из вашего приложения и автоматический запуск игры при вставке компакт-диска в дисковод. DirectSetup представляет собой небольшой набор функций, которые загружают файлы DirectX на компьютер пользователя во время запуска вашей программы и заносят все необходимые записи в системный реестр. AutoPlay — это обычная подсистема для работы с компакт-дисками, которая ищет в корневом каталоге компакт-диска файл `Autoplay.inf`. Если этот файл обнаружен, то AutoPlay запускает команды из этого файла.

**DirectX Graphics.** Компонент, в котором Microsoft решила объединить возможности DirectDraw и Direct3D, чтобы увеличить производительность и сделать доступными трехмерные эффекты в двумерной среде. Однако, на мой взгляд, не стоило отказываться от DirectDraw, и не только **потому**, что многие программы используют его. Использование **Direct3D** для получения двумерной графики в большинстве случаев неудобно и громоздко. Во многих программах, которые по своей природе являются двумерными приложениями (такие, как **GUI-приложения** или простейшие игры), использование **Direct3D** явно избыточно. Однако не стоит беспокоиться об этом, так как мы будем использовать интерфейс DirectX 7.0 для работы с DirectDraw.

**DirectX Audio.** Результат слияния **DirectSound** и DirectMusic, далеко не такой фатальный, как DirectX Graphics. Хотя данное объединение и более тесное, чем в случае с DirectX Graphics, но при этом из DirectX ничего не было удалено. В DirectX 7.0 DirectMusic был полностью основан на COM и мало что делал самостоятельно, а кроме того, он не был доступен из DirectSound. Благодаря DirectX Audio ситуация изменилась, и у вас теперь есть возможность работать одновременно и с DirectSound и с DirectMusic.

**DirectShow.** Компонент для работы с **медиапотоками** в среде Windows. **DirectShow** обеспечивает захват и воспроизведение мультимедийных потоков. Он поддерживает широкий спектр форматов— Advanced Streaming Format (ASF), Motion Picture Experts Group (MPEG), **Audio-Video Interleaved (AVI)**, MPEG Audio **Layer-3 (MP3)**, а также WAV-файлы. DirectShow **поддерживает** захват с использованием устройств Windows Driver Model (WDM), а также более старых устройств Video for Windows. Этот компонент тесно интегрирован с другими технологиями **DirectX**. Он автоматически определяет наличие аппаратного ускорения и использует его, но в состоянии работать и с системами, в устройствах которых аппаратное ускорение отсутствует. Это во многом облегчает вашу задачу, так как раньше, чтобы использовать в игре видео, вам приходилось либо использовать библиотеки сторонних разработчиков, либо самому писать эти библиотеки. Теперь же все это уже реализовано в DirectShow. **Сложность** лишь в том, что это довольно сложная система, требующая достаточно много **времени**, чтобы разобраться в ней и научиться ею пользоваться.

У вас, наверное, возник вопрос: как же разобраться в этих компонентах и версиях DirectX? Ведь все может **стать** совсем другим за какие-то полгода. Отчасти это так. Бизнес, которым мы занимаемся, очень рискованный — технологии, связанные с графикой и играми, меняются очень быстро. Однако, так как DirectX основан на технологии COM, программы, написанные, скажем, для DirectX 3.0, обязательно будут работать и с DirectX 8.0. Давайте посмотрим, как это получается.

## Краткое введение в COM

COM была задумана много лет назад как белый лист бумаги на базе новой парадигмы **программирования**, основной принцип которой можно сравнить с **принципом** конструктора: вы просто соединяете отдельные части и в результате получаете работающее единое **целое**. Каждая плата в компьютере (или каждый блок конструктора Lego) точно знает, как она должна функционировать, благодаря чему функционирует и собранный из них компьютер (или собранная из деталей игрушечная машина). Чтобы реализовать такую же технологию в программировании, необходим интерфейс общего назначения, который может представлять **собой** множество разнообразных функций любого типа. Именно в этом и состоит суть COM.

При работе с платами компьютера наиболее приятно то, что, добавляя новую штату, вы не должны отчитываться об этом перед другими платами. Как вы понимаете, в случае программного обеспечения **ситуация** несколько сложнее. Программу придется, по крайней мере, перекомпилировать. И это вторая проблема, решить которую призвана COM.

Нам требуется возможность расширять возможности **COM-объектов** так, чтобы программы, работавшие со старой версией объекта, продолжали успешно работать и с новой версией. Кроме **того**, можно изменить **COM-объекты**, не перекомпилируя при этом саму программу, — а это очень большое **преимущество**.

Поскольку вы можете заменять старые **COM-объекты** новыми, не перекомпилируя при этом прогамму, вы можете обновлять ваше программное обеспечение на машине **пользователя**, не создавая никаких очередных заплат или новых версий прогамм. Например, у вас есть программа, которая использует три **COM-объекта**: один отвечает за графику, один за звук и один за **работу** в сети (рис. 2.10). А теперь представьте, что вы уже продали 100000 копий своей программы и хотите избежать отсылки 100000 новых версий. При использовании **COM** для обновления работы с графикой вам достаточно дать своим пользователям новый **COM-объект**, и старая прогамма будет использовать его вместо старого; при этом не нужно делать решительно ничего: ни перекомпилировать, ни компоновать приложение. Конечно, **реализация** такой технологии на уровне **программирования** — задача очень сложная. Чтобы создать собственный **COM-объект**, требуется приложить массу усилий. Но зато как легко будет потом им пользоваться!

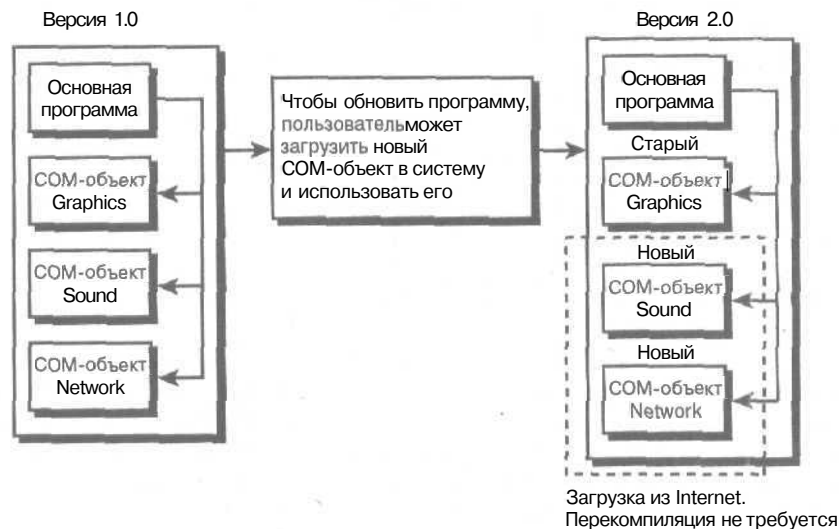


Рис. 2.10. Представление о **COM**

Следующий вопрос заключается в том, как и в каком виде распространять **COM-объекты** с учетом их природы **Plug and Play**. Четких правил на этот счет не существует. В большинстве случаев **COM-объекты** представляют собой динамически компокуемые библиотеки (**DLL**), которые могут поставляться вместе с прогаммой или **загружаться** отдельно. Таким образом, **COM-объекты** могут быть легко обновлены и изменены. Проблема лишь в том, что прогамма, использующая **COM-объект**, должна уметь загрузить его из **DLL**. Позже мы вернемся к этому вопросу.

## Что такое **COM-объект**?

В действительности **COM-объект** представляет собой класс (или набор классов) на языке **C++**, который реализует ряд **интерфейсов** (которые, в свою очередь, являются наборами функций). Эти интерфейсы используются для связи с **COM-объектами**. Взгля-

ните на рис. 2.11. На нем вы видите простой COM-объект с тремя интерфейсами: IGRAPHICS, ISOUND и IINPUT.

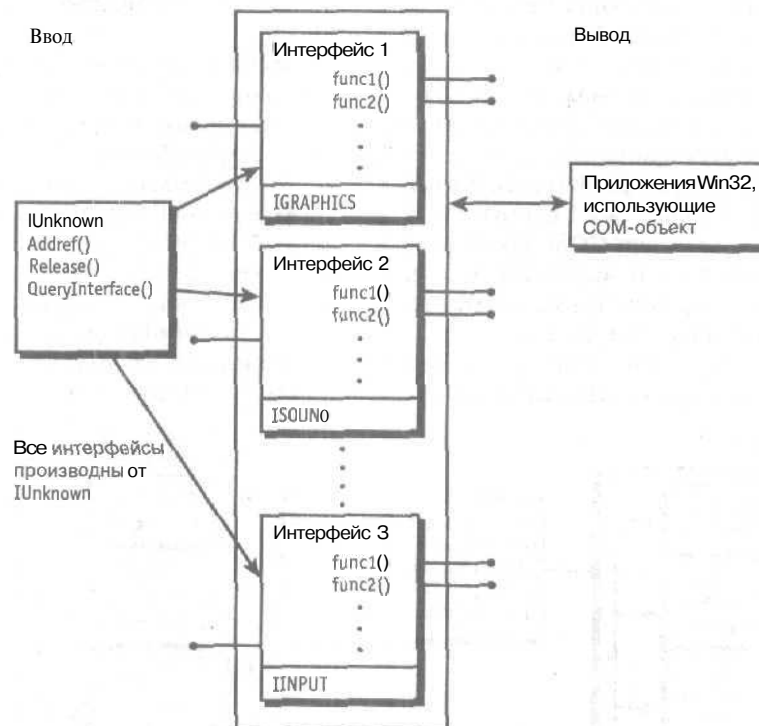


Рис. 2.11. Интерфейсы COM-объекта

У каждого из этих трех интерфейсов есть свой набор функций, который вы можете (если знаете, как) использовать в своей работе. Итак, каждый COM-объект может иметь один или несколько интерфейсов и у вас может быть один или несколько COM-объектов. В соответствии со спецификацией COM, все интерфейсы, созданные вами, должны быть производными от специального базового класса IUnknown. Для вас, как для программиста на C++, это означает, что IUnknown — это некая отправная точка, с которой нужно начинать создание интерфейсов.

Давайте взглянем на определение класса IUnknown.

```
struct IUnknown
{
    // Эта функция используется для получения
    // указателей на другие интерфейсы
    virtual HRESULT __stdcall QueryInterface(
        const IID &iid, (void **)ip) = 0;

    // Это функция увеличения счетчика ссылок
    virtual ULONG __stdcall AddRef() = 0;

    // Это функция уменьшения счетчика ссылок
    virtual ULONG __stdcall Release() = 0;
};
```

Обратите внимание на то, что все методы виртуальны и не имеют тела (абстрактны). К тому же все методы используют соглашение `_stdcall`, в отличие от привычных вызовов в C/C++. При использовании этого соглашения аргументы функции вносятся в стек справа налево.

Определение этого класса выглядит немного причудливо, особенно если вы не привыкли к использованию виртуальных функций. Давайте внимательно проанализируем `IUnknown`. Итак, все интерфейсы, наследуемые от `IUnknown`, должны иметь, по крайней мере, следующие методы: `QueryInterface()`, `AddRef()`, `Release()`.

Метод `QueryInterface()` — это ключевой метод COM. С его помощью вы можете получить указатель на требующийся вам интерфейс. Для этого нужно знать *идентификатор интерфейса*, т.е. некоторый уникальный код этого интерфейса. Это число длиной 128 бит, которое вы назначаете вашему интерфейсу. Существует  $2^{128}$  возможных значений идентификатора интерфейса, и я гарантирую, что нам не хватит и миллиарда лет, чтобы использовать их все, даже если на Земле все только и будут делать, что днем и ночью создавать COM-объекты! Несколько позже в этой главе, когда будут рассматриваться реальные примеры, мы еще затронем тему идентификаторов интерфейсов.

Кроме того, одно из правил COM гласит: если у вас есть интерфейс, то вы можете получить доступ к любому другому интерфейсу, так как они все происходят из одного COM-объекта. В общих чертах это означает: где бы вы ни находились, вы можете попасть куда захотите (рис. 2.12).

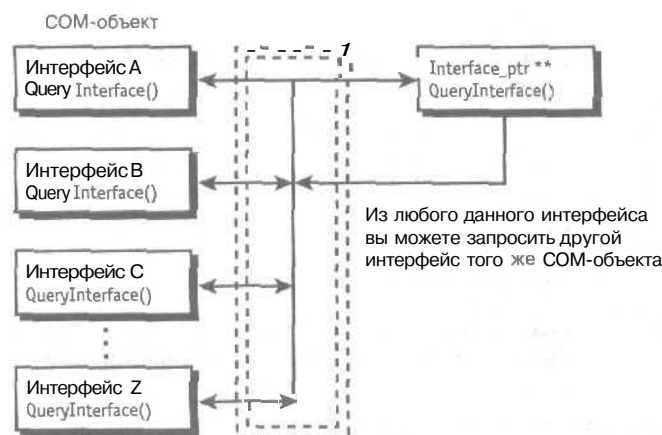


Рис. 2.12. Работа с интерфейсами COM-объекта

Довольно любопытна функция `AddRef()`. В COM-объектах используется технология, называемая счетчиком ссылок (reference counting) и отслеживающая все ссылки на объекты. Необходимость в этом объясняется одной из особенностей COM-технологии: она не ориентируется на конкретный язык программирования. Следовательно, когда создается COM-объект или интерфейс, для того чтобы отследить, сколько имеется ссылок на этот объект, вызывается `AddRef()`. Если бы COM-объект использовал вызовы `malloc()` или `new[]` — это было бы явным требованием использовать для разработки конкретный язык программирования C или C++. Когда счетчик ссылок обнуляется, объект автоматически уничтожается.

Тут же возникает второй вопрос: если COM-объекты — это классы, написанные на C++, то каким образом их можно создавать или использовать в Visual Basic, Java, ActiveX и т.д.? Просто так уж сложилось, что создатели для реализации COM использовали вир-

туальные классы C++. Вас никто не заставляет использовать именно этот язык программирования. Главное, чтобы созданный вами машинный код был аналогичен создаваемому компилятором Microsoft C++ при создании виртуальных классов. Тогда созданный COM-объект будет вполне корректен и работоспособен.

**СОВЕТ**

Функция `AddRef()` интерфейсов или COM-объектов вызывается автоматически функцией `QueryInterface()`, т.е. вызывать ее самостоятельно не нужно. Но вы можете захотеть вызывать ее явно: например, если по какой-либо причине пожелаете увеличить счетчик ссылок на объект, чтобы этот объект считал, что число указателей, которые указывают на него, больше, чем на самом деле.

У большинства компиляторов есть специальные дополнительные инструменты и возможности для создания COM-объектов, так что особой проблемы эта задача не представляет. Самое замечательное во всем этом то, что вы можете создавать COM-объекты в C++, Visual Basic или Delphi и затем эти объекты могут быть использованы любым из перечисленных языков. Бинарный код есть бинарный код!

Функция `Release()` служит для уменьшения счетчика ссылок COM-объекта или интерфейса на единицу. В большинстве случаев эту функцию вам придется вызывать самостоятельно по окончании работы с интерфейсом. Однако если вы создаете один объект из другого, то вызов функции `Release()` родительского объекта автоматически вызовет `Release()` дочернего объекта. Тем не менее, лучше все же самостоятельно вызывать `Release()` в порядке, обратном порядку создания объектов.

## Создание и использование COM-интерфейсов DirectX

Я думаю, теперь вам понятно, что COM-объекты — это набор интерфейсов, которые представляют собой не что иное, как указатели на функции (или, точнее, таблицу виртуальных функций). Следовательно, все, что вам нужно для работы с COM-объектами DirectX, — это создать объект, получить указатель на интерфейс, а затем обращаться к интерфейсу с использованием соответствующего синтаксиса. В качестве примера я буду использовать основной интерфейс `DirectDraw`.

Прежде всего, чтобы экспериментировать с `DirectDraw`, нам нужно следующее.

- Должны быть загружены и зарегистрированы COM-объекты времени исполнения `DirectDraw` и соответствующие динамические библиотеки. Все это делает инсталлятор `DirectX`.
- Необходимо включить в Win32-программу библиотеку импорта `DDRAW.LIB`, в которой находятся функции для работы с COM-объектами.
- Необходимо подключить к программе заголовочный файл `DDRAW.H`, чтобы компилятор мог получить информацию о типах данных и прототипах функций, используемых при работе с `DirectDraw`.

С учетом всего сказанного, вот как будет выглядеть тип данных для указателя на интерфейс для разных версий `DirectDraw`. В случае `DirectDraw 1.0` это `LPDIRECTDRAW` `lpdd = NULL`; для `DirectDraw 4.0` — `LPDIRECTDRAW4` `lpdd = NULL`; для `DirectDraw 7.0` — `LPDIRECTDRAW7` `lpdd = NULL`. Версии `DirectDraw 8.0`, как уже отмечалось, не существует.

Теперь, чтобы создать COM-объект `DirectDraw` и получить указатель на интерфейс объекта `DirectDraw` (который представляет видеокарту), достаточно использовать функцию `DirectDrawCreate()`.

```
DirectDrawCreate(NULL, &lpdd, NULL);
```

Не беспокойтесь о параметрах. Если вам это интересно, обратитесь к справочной системе DirectX SDK, однако в 99 случаях из 100 функция будет выглядеть так же. Т.е. используется только один параметр, и это — адрес указателя интерфейса, заполняемый адресом DirectDraw COM-интерфейса. Сейчас просто поверьте, что этот вызов создает объект DirectDraw и устанавливает указатель интерфейса на lpdd.

**НА ЗАМЕТКУ**

Конечно, в этой функции выполняется масса вещей. Она открывает .DLL, загружает ее, делает вызовы, а также много чего еще, но вам не нужно беспокоиться об этом.

После этого можно обращаться к DirectDraw. Хотя нет — ведь вы же не знаете, какие методы и функции вам доступны, именно поэтому вы и читаете эту книгу! В качестве примера установим видеорежим 640x480 с 256 цветами.

```
lpdd->SetVideoMode(640, 480, 256);
```

Просто, не правда ли? Единственная дополнительная работа, которая при этом должна быть выполнена, — это разыменование указателя на интерфейс lpdd. Конечно, на самом деле происходит поиск в виртуальной таблице интерфейса, но не будем вдаваться в детали.

По существу, любой вызов DirectX выполняется следующим образом.

```
interface_pointer->method_name(parameter list);
```

Непосредственно из интерфейса DirectDraw вы можете получить и другие интерфейсы, с которыми предстоит работать (например Direct3D); для этого следует воспользоваться функцией `QueryInterface()`.

## Запрос интерфейсов

Как ни странно, но в DirectX номера версий не синхронизированы, и это создает определенные проблемы. Когда вышла первая версия DirectX, интерфейс DirectDraw назывался IDirectDraw. Когда вышла вторая версия DirectX, DirectDraw был обновлен до версии 2.0 и мы получили интерфейсы IDirectDraw и IDirectDraw2. В версии 6.0 набор интерфейсов расширился до IDirectDraw, IDirectDraw2 и IDirectDraw4. После выхода версии 7.0 к ним добавился интерфейс IDirectDraw7, который остается последним, так как в версии DirectX 8.0 DirectDraw не поддерживается.

Вы спрашиваете, что случилось с третьей и пятой версией? Понятия не имею! Но в результате, даже если вы используете DirectX 8.0, это еще не означает, что все интерфейсы обновлены до этого же номера версии. Более того, разные интерфейсы могут иметь разные версии. Например, DirectX 6.0 может иметь интерфейс DirectDraw версии 4.0 — IDirectDraw4, но DirectSound при этом имеет версию 1.0 и его интерфейс имеет имя IDirectSound. Кошмар! Мораль сей басни такова: когда вы используете интерфейс DirectX, вы должны убедиться в том, используется ли самая последняя его версия. Если вы в этом не уверены, воспользуйтесь указателем на интерфейс версии 1.0, получаемый при создании объекта, чтобы получить указатель на интерфейс последней версии.

Непонятно? Вот конкретный пример. `DirectDrawCreate()` возвращает указатель на базовый интерфейс первой версии, но нас интересует интерфейс DirectDraw под именем IDirectDraw7. Как же нам получить все возможности интерфейса последней версии?

Существует два пути: использовать низкоуровневые функции COM или получить указатель на интересующий нас интерфейс при помощи вызова `QueryInterface()`, что мы сейчас и сделаем. Порядок действий следующий: сначала создается COM-интерфейс DirectDraw с помощью вызова `DirectDrawCreate()`. При этом мы получаем указатель на интерфейс IDirectDraw. Затем, используя этот указатель, мы вызываем `QueryInterface()`,

передавая ему идентификатор **интересующего** нас интерфейса, и получаем указатель на него. Вот как выглядит соответствующий код.

```
LPDIRECTDRAW lpdd; // версия 1.0
LPDIRECTDRAW7 lpdd7; // версия 7.0
// Создаем интерфейс объекта DirectDraw1.0
DirectDrawCreate (NULL, &lpdd, NULL);

// В DDRAW.H находим идентификатор интерфейса IDirectDraw7
// и используем его для получения указателя на интерфейс
lpdd->QueryInterface(IID_IDirectDraw7, &lpdd7);

// Теперь у вас имеется два указателя. Так как указатель
// на IDirectDraw нам не нужен, удалим его.
lpdd->Release();

// Присвоение значения NULL для безопасности
lpdd=NULL;

// Работа с интерфейсом IDirectDraw7

К По окончании работы с интерфейсом мы должны удалить его
lpdd7->Release();
// Присвоение значения NULL для безопасности
lpdd7=NULL;
```

Теперь вы знаете, как получить указатель на один интерфейс с помощью другого. В DirectX 7.0 Microsoft добавила новую функцию **DirectDrawCreateEx()**, которая сразу возвращает интерфейс IDirectDraw7 (и после этого тут же, в DirectX 8.0, Microsoft полностью отказывается от **DirectDraw...**).

```
HRESULT WINAPI DirectDrawCreateEx(
    GUID FAR *lpGUID, // GUID для драйвера,
                      // NULL для активного дисплея
    LPVOID *lpIdd, // Указатель на интерфейс
    REFIID iid, // ID запрашиваемого интерфейса
    IUnknown FAR *pUnkOuter // Расширенный COM. NULL
);
```

При вызове этой функции вы можете сразу указать в iid требующуюся версию DirectDraw. Таким образом, вместо описанных выше вызовов **QueryInterface()** мы просто вызываем функцию **DirectDrawCreateEx()**.

```
LPDIRECTDRAW7 lpdd7; // Версия 7.0
// Создание интерфейса объекта DirectDraw 7.0
DirectDrawCreateEx(NULL, (void*)&lpdd7, IID_IDirectDraw7, NULL);
```

Вызов **DirectDrawCreateEx()** создает запрошенный интерфейс, и вам не придется использовать для этого DirectDraw 1.0. Вот и все, что касается принципов использования DirectX и COM. Теперь дело за изучением сотен функций и интерфейсов DirectX.

## Резюме

В данной главе даются сведения по основам программирования для Windows, а также о том, что представляет собой DirectX и как он связан с Windows. Замечательной

особенностью этого материала является то, что вам не нужно слишком много знать о Windows или DirectX (или обо всех их версиях), чтобы получить массу полезной информации из этой книги, поскольку в следующей главе мы собираемся построить виртуальный компьютер, на котором будем проводить свои трехмерные эксперименты и писать программный код. Тем самым вы будете изолированы от внутреннего механизма Win32/DirectX и сможете сосредоточиться на материале по трехмерной графике. Конечно, знания по Win32/DirectX вам не мешают, поэтому если вы хотите получить дополнительную информацию, почитайте книгу *Программирование игр для Windows. Советы профессионала*, а также какую-нибудь хорошую книгу по программированию для Windows.



# ГЛАВА 3

## Виртуальный компьютер для программирования трехмерных игр

### В этой главе...

• Введение в интерфейс виртуального компьютера	120
• Построение интерфейса виртуального компьютера	122
• Консоль игры T3DLIB	132
• Библиотека T3DLIB1	138
• Система ввода DirectX	182
• Звуковая и музыкальная библиотека T3DLIB3	188
• Окончательная версия консоли игры	197
• Образцы приложений T3DLIB	211

В этой главе мы в основном займемся созданием виртуальной компьютерной системы, основанной на программном интерфейсе. Она будет поддерживать линейные 8- и 16-битовые буферы кадров (двойная буферизация) и входные устройства, а также возможность работы с музыкой и звуком. Имея такой интерфейс, мы сможем: в оставшейся части книги сосредоточиться на проблемах трехмерной математики, графики и программирования игр, не отвлекаясь на ~~всякие~~ мелочи типа получения ввода от клавиатуры. Вот о чем мы поговорим в данной главе:

- разработка графического интерфейса виртуального компьютера;
- построение консоли игры для Windows;

- список функций **API** библиотеки T3DLIB из книги *Программирование игр для Windows. Советы профессионала*;
- реализация виртуального компьютера при помощи библиотеки T3DLIB;
- окончательная версия консоли игры;
- использование игровой библиотеки T3DLIB.

## Введение в интерфейс виртуального компьютера

Цель данной книги — научить читателя работать с трехмерной графикой и программировать игры. Однако передо мной как автором стоит дилемма — на что в первую очередь должна быть ориентирована данная книга? Только ли на графику и игры — без упоминания таких вещей, как детали программирования для Win32, DirectX и т.п.? Несмотря на то, что после прочтения предыдущей главы у вас должны появиться определенные знания о программировании Win32 и DirectX, этого все же слишком мало. Поэтому мое решение вопроса состоит в создании "виртуального компьютера" на основе разработанного в предыдущей книге (*Программирование игр для Windows. Советы профессионала*) игрового процессора, который вы будете использовать как "черный ящик", так что мы сможем уделить больше внимания вопросам трехмерной графики и программирования игр.

### НА ЗАМЕТКУ

Это подход, используемый OpenGL. Вы используете библиотеки OpenGL, которые отвечают за низкоуровневую работу с устройствами и выделение и освобождение ресурсов, включая такие вещи, как открытие окон и получение пользовательского ввода.

Такой подход вполне обоснован по целому ряду причин. Ведь нас в первую очередь интересует программирование игры, а не частности, связанные с получением ввода, выводом музыкального сопровождения или музыкальных эффектов. 99% нашей работы связано с растеризацией, отображением текстур, освещением, удалением скрытых поверхностей и т.п. Что нам реально надо — это пустой экран, возможность получить информацию от джойстика, мыши и клавиатуры и, возможно, вывод музыки и звуковых эффектов.

Вместо того, чтобы копаться в деталях Win32, разбираться с DirectX и т.п., мы можем просто воспользоваться готовым API, который был разработан в предыдущей книге, в качестве инструмента для создания обобщенного виртуального компьютера, с помощью которого будут проведены наши эксперименты в области трехмерной графики и написаны наши игры.

Использование такого уровня косвенности приводит к тому, что разрабатываемый код оказывается достаточно обобщенным для того, чтобы быть легко перенесенным на другую платформу, например, для работы под управлением Mac или Linux. Все, что потребуется для этого — это эмулировать низкоуровневые интерфейсы, такие как графическая система с двойной буферизацией, вывод музыки, звука, и работа с устройствами ввода. Весь прочий код будет одним и тем же на всех платформах.

Единственный недостаток игрового процессора из первой книги состоит в том, что множество структур данных, да и сам дизайн оказались определяемыми DirectX. Это связано с тем, что в целях повышения производительности процессора я стремился сделать рассматриваемый уровень между процессором и DirectX как можно более тонким. Однако вполне можно разработать дополнительный уровень над разработанным мною, который будет полностью машинно-независимым (но я не думаю, что эта задача стоит необходимого для ее решения времени). Главное в том, что если вы хотите перенести игру на

другую платформу, то если вы сможете создать экран с двойной буферизацией, то все остальное будет нормально работать. Мои функции звукового сопровождения и пользовательского ввода не делают ничего, кроме вызова соответствующих функций DirectX.

Как только вы разберетесь с API, предоставляемым моим игровым процессором, вы можете забыть о Win32, DirectX и прочем — после этого вам нужно будет только корректно заполнить буфер кадра.

Итак, мы собираемся создать спецификацию виртуального, или абстрактного графического компьютера, а затем реализовать его при помощи API из предыдущей книги. Однако главное назначение виртуального компьютера — позволить нам вплотную заняться программированием игр. С учетом этого вот как выглядит список возможностей, которые нужны нам для написания трехмерной игры для произвольного компьютера.

1. Возможность создания окна или экрана с определенной глубиной цвета и линейной адресацией двумерной матрицы пикселей. Кроме того, нужна поддержка внеэкранной страницы или двойной буферизации, чтобы изображение могло быть визуализировано без вывода на экран а затем скопировано или переключено на основной экран для создания эффекта анимации.
2. Возможность получения пользовательского ввода из системных устройств ввода, таких как клавиатура, мышь и джойстик. Ввод должен быть в простом и легко обрабатываемом формате.
3. Возможность загрузки и воспроизведения звуковых эффектов и музыки в стандартных форматах типа .WAV или .MID. (Необязательно).

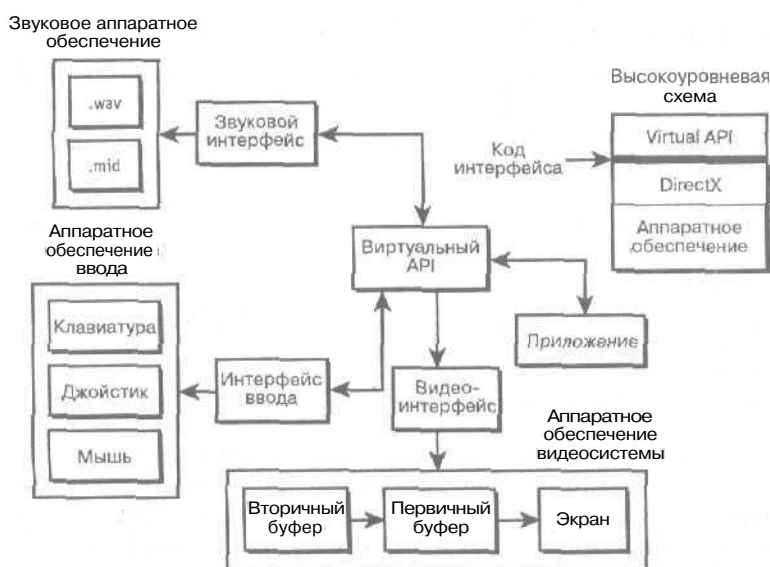


Рис. 3.1. Диаграмма системных уровней виртуального компьютера

На рис. 3.1 приведена схема создаваемого нами виртуального компьютера. Главный принцип, положенный в основу этого компьютера, — "если я могу вывести пиксель и прочесть нажатую клавишу, я в состоянии написать *Doom*". Это примерно так и есть — все, что вам надо, это доступ к буферу кадра, а все остальное решается просто. Кроме того, поскольку наша книга посвящена программированию трехмерных игр, ни один из рассматриваемых алгоритмов не будет использовать возможности аппаратного ускоре-

ния (это даже как-то не *спортивно...*). Мы сами будем чертить многоугольники, определять освещенность и выполнять все вычисления, *необходимые* для создания изображения в буфере кадра.

**НА ЗАМЕТКУ**

Вы можете спросить — зачем использовать программные решения, если доступны аппаратные? Во-первых, чтобы быть настоящим профессионалом, надо уметь решать такие задачи без "аппаратных костылей". Во-вторых, знание того, как это все работает, поможет вам при изучении аппаратных ускорителей. И в-третьих, кто, по-вашему, пишет код для аппаратного обеспечения?!

## Построение интерфейса виртуального компьютера

Построение интерфейса виртуального компьютера достаточно просто, если у нас есть функциональная библиотека, реализующая перечисленные в предыдущем разделе возможности, а она у нас есть — это библиотека из предыдущей книги. Перед тем как перейти к рассмотрению конкретных деталей, давайте сконструируем "модель" функций и структур данных, которые должны быть в интерфейсе виртуального компьютера. Конечно, это всего лишь пример, и пример очень высокоуровневый. К сожалению, окончательная версия виртуального компьютера будет несколько иной, чем рассматриваемая сейчас, поскольку при его реализации мы будем *вынуждены* углубиться в различные детали в большей степени.

### Буфер кадра и видеосистема

Перед тем как начать, будем считать, что у нас есть способ инициализации видеосистемы и открытия окна с определенным разрешением и глубиной цвета. Назовем соответствующую функцию `Create_Window()`. Вот как она может выглядеть.

```
Create_Window(int width, int height int bit_depth);
```

Для создания экрана размером 640x480 с глубиной цвета 8 битов используется *следующий* вызов.

```
Create_Window(640, 480, 8);
```

Соответственно, для создания экрана 800x600 с глубиной цвета 16 битов вызов будет таким.

```
Create_Window(800, 600, 16);
```

Не забывайте, что за таким простым вызовом в действительности может скрываться целое множество вызовов других функций — просто сейчас это нас не интересует. Да и само "окно" может быть как обычным окном Windows, так и всем дисплеем в полноэкранном режиме.

Как я говорил, мы заинтересованы в разработке системы, которая опирается на первичный, видимый буфер дисплея, и вторичный, *внеэкранный* буфер, который *является* невидимым. Оба буфера должны быть линейно адресуемы, причем одно *слово* (которое, в зависимости от глубины цвета, может быть BYTE, WORD или QUAD) представляет отдельный пиксель. Наша *система* буферов кадров представлена на рис. 3.2.

В буфере кадра (первичном или вторичном) имеется *область заполнения памяти*, т.е. в ряде видеосистем используется линейная адресация *к* пределам строки, но *между* строками имеется определенный скачок адресов. Пример такой видеосистемы приведен на рис. 3.3. Ее характеристики — 640x480x8, т.е. на один пиксель приходится один байт памяти, соответственно, на строку должно приходиться 640 байтов. Но, как видно из рисунка, на строку приходится большее количество байтов, что связано с адресацией памяти в видеокарте. Нам нужна соответствующая модель, которая учитывает данный *шаг памяти* (memory pitch).

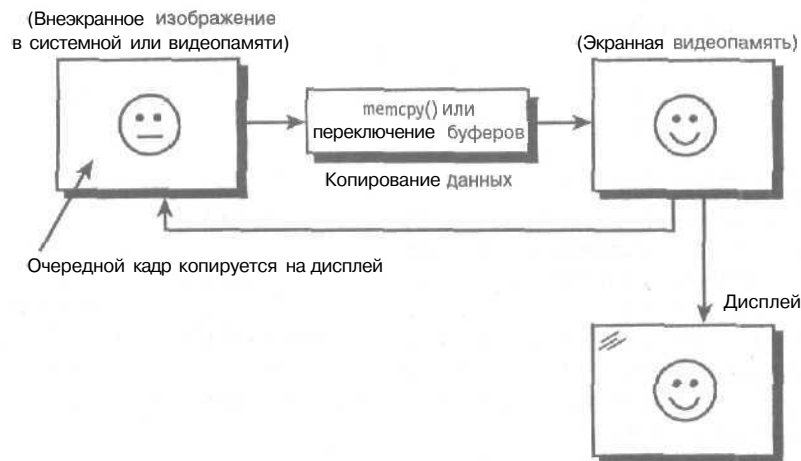


Рис. 3.2. Система буферов виртуального компьютера

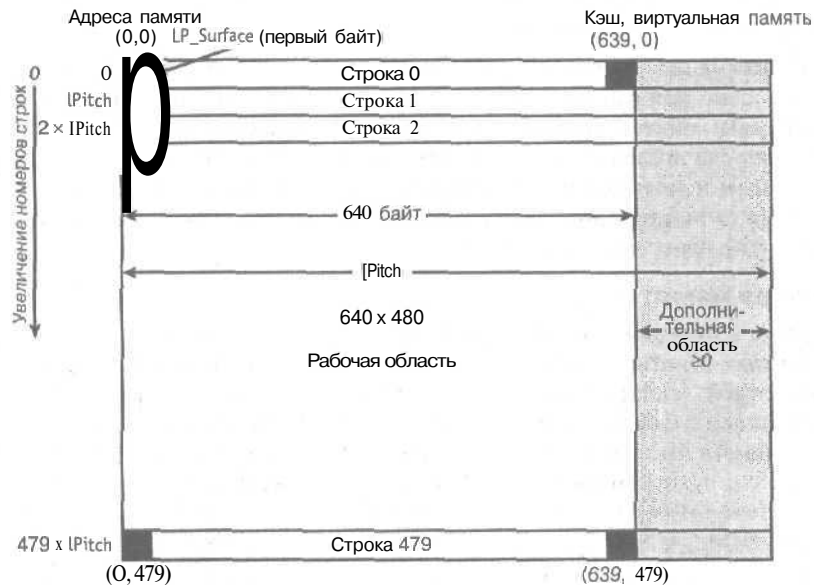


Рис. 3.3. Аппаратный буфер кадра

В такой модели учитывается, что видеокарта может иметь дополнительную область памяти за пределами строки, связанную с особенностями кэширования или аппаратной адресации. Такая организация видеопамяти не представляет никакой проблемы — мы просто должны учесть, что шаг памяти не совпадает с шагом видеопамяти. Так, чтобы обратиться к пикселю в системе с глубиной цвета 8 битов, мы используем *следующий* код.

```
UCHAR *video_buffer; // Указатель на видеобуфер
int x,y;             // Координаты пикселя
int memory_pitch;    // Количество памяти на строку

video_buffer[x+y*memory_pitch] = pixel;
```

В системе с глубиной цвета 16 битов код почти ничем не отличается от рассмотренного.

```
USHORT *video_buffer; // Указатель на видеобuffer
int x,y; // Координаты пикселя
int memory_pitch; // Количество памяти на строку
```

```
video_buffer[x+y*(memory_pitch >> 1)]= pixel;
```

Конечно, если переменная `memory_pitch` указывает количество слов `USHORT` на одну строку, то операция сдвига становится излишней.

С учетом сказанного, приступим к разработке модели первичного и вторичного буферов. Учтите, что это всего лишь модель, и ее детали могут измениться, когда мы приступим к ее реализации с использованием игрового процессора T3DLIB. В любом случае нам нужны указатели на первичный и вторичный буфер и переменная для хранения шага памяти. Таким образом, нам нужны следующие глобальные переменные.

```
UCHAR *primary_buffer; // Первичный буфер
int primary_pitch; // Шаг памяти буфера
```

```
UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch; // Шаг памяти буфера
```

Обратите внимание, что оба указателя имеют тип `UCHAR*`, и что шаг памяти выражен в байтах. Это удобно при работе с 8-битовой глубиной цвета, но при работе в 16-битовом режиме на каждый пиксель приходится по 2 бита. В этом случае вы можете при желании выполнить явное приведение типа к `USHORT*`, но мне лично представляется более логичным и понятным использование указателя `UCHAR*` и значения шага памяти в байтах. В этом случае все функции, которые нам предстоит написать, будут использовать одни и те же параметры, независимо от режима работы.

## Блокировка памяти

При работе с буферами мы сталкиваемся с еще одной особенностью — *блокировкой* (locking) *памяти*. Многие видеокарты имеют специальную видеопамять, которая может быть переносимой, кэшируемой и т.п. Это означает, что при обращении к памяти для чтения или записи в буфер, вы должны дать системе знать об этом, с тем чтобы не произошло случайного изменения памяти во время вашей работы с ней. По завершении работы вы разблокируете память, и система может продолжать работу с ней.

### СОВЕТ

Если вы — старый программист, то ощутите прилив ностальгических чувств — ведь в версиях 1.0–3.0 Windows вы были обязаны заниматься блокировкой памяти.

Все это с точки зрения программистов означает, что указатели `primary_buffer` и `secondary_buffer` корректны только на время блокировки. Причем нельзя гарантировать, что их значения останутся теми же при очередной блокировке. Например, при первой блокировке указатель на буфер может иметь значение `0x0FFFFFFC00000000`, а при следующей — `0x0FFFFFFD00000000`. Это связано с работой аппаратного обеспечения, которое может, например, просто перенести буфер в другое место, так что будьте внимательны! В любом случае процедура использования блокировки выглядит следующим образом.

1. Заблокировать интересующий нас буфер (первичный или вторичный) и получить стартовый адрес и шаг памяти.
2. Выполнить необходимые действия с видеопамью.
3. Разблокировать буфер.

Само собой разумеется, в 99% случаев вы будете выполнять последовательность блокировка— чтение/запись—разблокирование только для вторичного буфера, поскольку явно не захотите, чтобы пользователь видел, как вы вносите изменения в изображение,

Исходя из описанной особенности обращения к видеопамяти, разработаем новые функции для выполнения блокировки и разблокирования.

```
Lock_Primary(UCHAR **primary_buffer,
             int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);

Lock_Secondary(UCHAR **secondary_buffer,
               int *secondary_pitch);
Unlock_Secondary(UCHAR *secondary_buffer);
```

Для блокировки буфера вы вызываете функцию, передавая ей в качестве параметров адреса переменных для хранения адреса буфера и шага памяти. Функция блокирует поверхность и записывает фактические значения в переданные ей переменные, после чего вы можете использовать их — до тех пор, пока не разблокируете поверхность вызовом соответствующей функции. Просто, правда?

В качестве примера посмотрим, как будет выглядеть запись одного пикселя на экране размером 800x600 как в 8-битовом, так и в 16-битовом режимах. Вот как это выглядит при глубине цвета, равном 8 битам.

```
UCHAR *primary_buffer; // Первичный буфер
int primary_pitch; // Шаг памяти

UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch; // Шаг памяти

UCHAR pixel; // Записываемый пиксель
int x,y; // и его координаты

// Шаг 1: Создание окна
Create_Window(800, 600, 8);

// Шаг 1: Блокировка вторичной поверхности
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// Запись пикселя в центре экрана
secondary_buffer[x + y*secondary_pitch] = pixel;

// Разблокирование буфера
Unlock_Secondary(secondary_buffer);
```

Все очень просто. А вот тот же код, но для 16-битового режима.

```
UCHAR *primary_buffer; // Первичный буфер
int primary_pitch; // Шаг памяти

UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch; // Шаг памяти

USHORT pixel; // Записываемый пиксель
int x,y; // и его координаты
```

```
// Шаг 1: Создание окна
Create_Window(800, 600, 16);

// Шаг 1: Блокировка вторичной поверхности
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// Запись пикселя в центре экрана. Здесь надо не забывать о
// том, что указатель, полученный при блокировке, имеет тип
// UCHAR*, а не USHORT*, так что мы должны выполнить явное
// преобразование типа

USHORT *video_buffer=(USHORT *)secondary_buffer;
video_buffer[x+y*(secondary_pitch >> 1)]=pixel;

// Разблокирование буфера
Unlock_Secondary(secondary_buffer);
```

Обратите внимание на деление шага памяти на 2 при помощи сдвига вправо на один бит, поскольку при использовании арифметики указателей нам надо преобразовать шаг памяти в количество USHORT на строку. Конечно, имеется масса способов написать этот код иначе, но **главное** — идея, которую вы, надеюсь, прекрасно уловили.

## Работа с цветом

В первую очередь при разработке игр исходя из требований скорости мы планируем работать с 8-битовой глубиной цвета с использованием палитры и с 16-битовым RGB цветом. В первом случае на один пиксель приходится один байт памяти, а во втором — два байта. Однако кодирование цвета в этих **режимах** принципиально различается.

### 8-битовый режим

В 8-битовом режиме используется стандартная таблица поиска цветов (color lookup table — **CLUT**) с 256 записями, показанная на рис. 3.4. 16-битовый режим использует стандартное RGB-кодирование. При работе в 8-битовом режиме мы должны заполнить таблицу RGB-значениями для каждой записи в таблице (их номера — 0...255). Мы считаем, что имеется функция (или функции), предназначенная для чтения и записи данных в таблице, так что нам незачем об этом беспокоиться.

В 8-битовом режиме нет никаких неожиданностей — на один пиксель требуется один байт памяти, и значение этого байта представляет собой индекс в таблице цветов, каждая запись которой представляет собой триаду 8-битовых значений для каждого цветового канала (красный, зеленый, синий) — в сумме 24 бита на одну запись.

#### СОВЕТ

Некоторые **видеокарты** используют по 6 битов из каждых 8, другими словами, вместо 256 оттенков цветов они обеспечивают только 64. С программной точки зрения никаких отличий при этом нет.

### 16-битовый режим

Работа в 16-битовом режиме несколько **проще**, чем в 8-битовом. В этом случае нет никакой таблицы цветов, а каждый пиксель в действительности представляет собой объединение битовых **строк**. Формат пикселя при этом может быть 5.5.5 или 5.6.5 — т.е. по 5 битов на красный и синий цвета, и 5 или 6 — на зеленый **цвет** (см. рис. 3.5). Непонятно почему, но у многих людей этот формат вызывает непонимание — в первую очередь в том

плане, как создается 16-битовый цвет. Все очень просто — надо только выполнить определенные действия с битами.

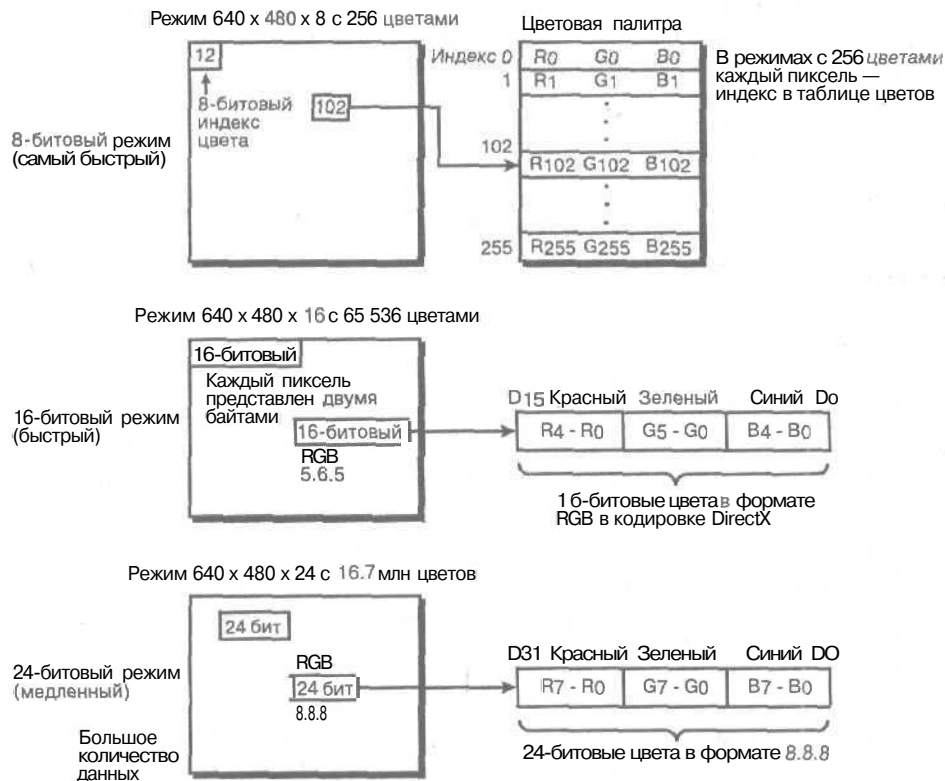


Рис. 3.4. Сравнение различных глубин цвета и их реализация



Рис. 3.5. 16-битовый цвет: форматы 5.6.5 и 5.5.5

В режиме 5.5.5 для каждого канала отведено по 5 битов, т.е. используется по 32 оттенка каждого из основных **цветов**, а значит, всего имеется  $32 \cdot 32 \cdot 32 = 32768$  различных **цветов**. Для того, чтобы преобразовать значения интенсивности каналов *r*, *g* и *b*, каждое в диапазоне от 0 до 31 (31 соответствует максимальной интенсивности), в RGB-цвет формата 5.5.5, можно использовать следующий макрос.

```
// Построение 16-битового цвета формата 5.5.5
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + \
((r & 31) << 10))
```

В формате 5.6.5 у зеленого цвету отведено на один бит больше, так что его интенсивность лежит в пределах от 0 до 63, что дает в сумме  $32 \cdot 64 \cdot 32 = 65536$  различных цветов. Вот макрос, используемый для создания цвета в данном формате.

```
// Построение 16-битового цвета формата 5.6.5
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + \
((r & 31) << 11))
```

Пока изложенного материала нам вполне достаточно, но в дальнейшем в реальной библиотеке при выборе 16-битового режима будет **определяться**, какой формат используется в данной системе, и будут устанавливаться соответствующие флаги и указатели функций, чтобы нам не приходилось **самостоятельно** разбираться с этими форматами.

## Анимация

Последняя функциональная возможность, необходимая нам, — реализация переключения буферов (или копирования вторичного буфера в первичный), как показано на рис. 3.6. Конечно, мы можем просто заблокировать оба буфера и **построчно** скопировать содержимое одного из них в другой. Однако множество видеокарт имеет специальные аппаратные возможности для переключения буферов и копирования их содержимого — гораздо быстрее, чем при помощи процессора. Очевидно, что данная возможность должна использоваться в нашем API. Назовем соответствующую функцию `Flip_Display()`. Она не получает никаких параметров и просто копирует вторичный буфер в первичный (программно или аппаратно, но мы не знаем, как именно решается эта задача в нашем "черном ящике").

### ВНИМАНИЕ

Вы можете удивиться, почему я говорю о построчном копировании вторичного буфера, а не при помощи функции `memcpy()`. Это связано с тем, что вторичный буфер может не быть непрерывным, так что копирование содержимого буфера надо проводить построчно, с учетом шага памяти.

Здесь есть одно **правило**, которому мы должны строго следовать: при вызове функции `Flip_Display()` как первичный, так и вторичный буферы должны быть разблокированы. Если это не так, переключение не произойдет и будет сгенерирована ошибка (если, конечно, вам не придется просто нажать кнопку `Reset...`). Всегда помните об этом — перед переключением буферов необходимо убедиться в том, что они разблокированы.

Кроме того, имеются две вспомогательные функции, которые могут пригодиться при анимации — функции очистки буферов. По тем же причинам, по которым нельзя использовать функцию `memcpy()` для копирования буферов, нельзя использовать функцию `memset()` для их очистки. Наши вспомогательные функции

```
Fill_Primary(int color);
Fill_Secondary(int color);
```

очищают буферы, используя при этом аппаратные возможности, если таковые имеются в наличии.

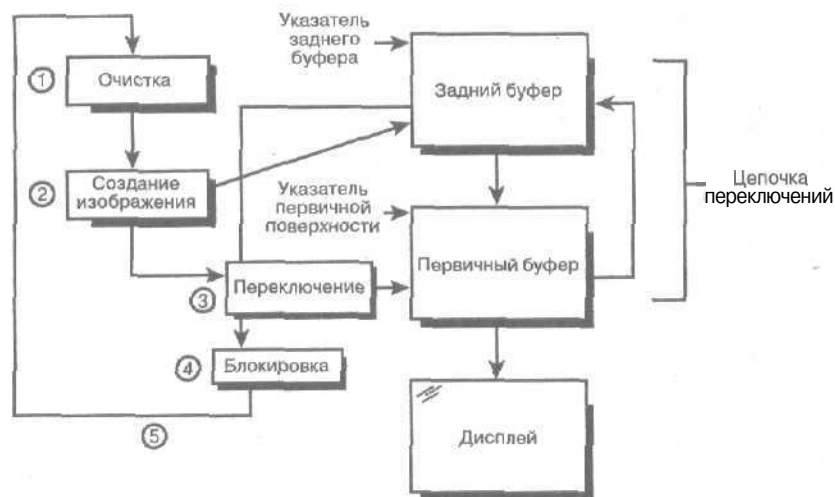


Рис. 3.6. Система анимации, основанная на переключении страниц

Данные функции предполагают, что буферы разблокированы, а параметр `color` в зависимости от используемого режима является 8-битовым или 16-битовым. В большинстве случаев очистка буфера заполняет его черным цветом, т.е. значение параметра — 0 (так как 0 в 16-битовом режиме соответствует черному цвету `RGB(0,0,0)`, а в 8-битовом режиме нулевая запись палитры, как правило, содержит черный цвет).

Использование этих функций очень простое. В начале анимационного цикла вы просто очищаете экран. Однако, поскольку позже вы копируете вторичный буфер в первичный, очищать последний нет необходимости, так что вам достаточно перед началом рисования очищать только вторичный буфер.

Вот пример анимации, выводящей на экран случайные точки в режиме 800x600x8 с частотой 30 fps.

```

UCHAR *primary_buffer; // Первичный буфер
int primary_pitch; // Шаг памяти в байтах

UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch; // Шаг памяти в байтах

UCHAR pixel; // Выводимый пиксель (8 битов)
int x,y; // Координаты пикселя

// Шаг 0: создание окна
Create_Window(800, 600, 8);

// Вход в бесконечный цикл
while(1)
{
    // Очистка вторичного буфера
    Fill_Secondary(0);

    // Шаг 1: блокировка вторичного буфера
  
```

```

Lock_Secondary(&secondary_buffer, &secondary_pitch);

for (int num_dots=0; num_dots < 1000; num_dots++)
{
    //Случайный пиксель
    x = rand()%800;
    y = rand()%600;
    pixel = rand()%256;

    // Вывод пикселя
    video_buffer[x+y*secondary_pitch] = pixel;
} // for num_dots

//Разблокируем вторичный буфер
Unlock_Secondary(secondary_buffer);

// Переключение буферов
Flip_Display();

// Синхронизирующее ожидание
Sleep(33);

} // while

```

Понятно, что в реальном приложении бесконечного цикла не будет, но здесь главное — понять саму идею.

## Полная виртуальная графическая система

В принципе, рассмотренной функциональности достаточно для реализации полноценной виртуальной графической системы, так что просто вкратце подведем итоги.

- Мы предполагаем наличие функций, которые позволяют настраивать графическую систему и открывать окно (частным случаем которого может быть весь экран) заданного размера и глубины цвета. В большинстве случаев из-за необходимости высокой скорости работы мы имеем дело с 8- и 16-битовым режимами.
- Система содержит два видеобuffers — первичный и внеэкранный **вторичный**. Буферы линейно адресуемы, однако могут не быть непрерывными и характеризуются определенной величиной шага памяти. Кроме того, при **обращении** к буферу его необходимо заблокировать, а после выполнения необходимых операций — разблокировать.
- Для анимации и переключения между страницами предполагается наличие функций, **использующих** возможности аппаратного ускорения блиттинга (blitting — от blocking image transfer). Перед вызовом таких функций мы должны убедиться в **разблокированности** первичного и вторичного буферов. Кроме того, имеются функции очистки **буферов**, использующие аппаратные возможности (при наличии таковых).

Как видите, если мы сможем реализовать эту виртуальную программную систему, у нас останется единственная проблема — трехмерная графика, которой мы и намерены заниматься. Все необходимое для вывода графики на экран у нас уже есть.

## Ввод-вывод, звук и музыка

Поскольку главное в данной книге — это создание трехмерных игр, нам совершенно необходимо получение ввода от игрока, а также возможность вывода звуковых эффектов и музыки. Иначе игрокам придется управлять игрой телепатически, имитируя звуки восприятия вроде “Пиф! Паф!”... :-)

Конечно, вся необходимая функциональность имеется в DirectX, и надо только написать соответствующие функции-оболочки. Главное, чего я хочу добиться, — чтобы при переносе на другую платформу, где не поддерживается DirectX, потребовалась минимальная работа. Игровой процессор из предыдущей книги обладает следующими фундаментальными возможностями.

1. Инициализация и обнаружение всех устройств ввода; клавиатуры, мыши, джойстика.
2. Считывание данных или *состояния* всех устройств ввода и **размещение** полученной информации в простых структурах данных.
3. Инициализация системы воспроизведения звуковых эффектов и музыки.
4. Загрузка с диска и воспроизведение (одноразовое или циклическое) звуковых эффектов в формате .WAV и музыки в формате MIDI. Возможность проверки состояния звука и музыки в процессе воспроизведения.
5. Корректное выключение всех подсистем.

Здесь перечислена вся **функциональность**, которая будет реализована в виртуальном компьютере — не больше, но и не меньше.

Вот как должны выглядеть функции, реализующие описанные возможности.

```
// Инициализация всех устройств ввода
Init_Input_Devs();

// Завершение работы системы ввода
Shutdown_Input_Devs();

// Чтение ввода с клавиатуры; data — некоторая структура для
// хранения состояния клавиатуры (возможно, представляет
// собой массив)
Read_Keyboard(&data);

// Чтение данных мыши; data — некоторая структура для
// хранения состояния мыши — положения и состояния кнопок
Read_Mouse(&data);

// Чтение данных джойстика; data — некоторая структура для
// хранения состояния джойстика — положения и состояния
// кнопок
Read_Joystick(&data);

// Инициализация системы работы со звуком и музыкой
Init_Sound_Music();

// завершение работы системы воспроизведения звука и музыки
Shutdown_Sound_Music();

// Загрузка .WAV-файла и возврат его идентификатора
int id = Load_WAV(char *filename);
```

```
// Воспроизведение .WAV-файла по его идентификатору
Play_WAV(int id);

// Загрузка .MID-файла и возврат его идентификатора
int id = Load_MID(char *filename);

// Воспроизведение .MID-файла по его идентификатору
Play_MID(int id);
```

Если мы реализуем перечисленные функции, то у нас будет все необходимое для работы с устройствами ввода и звуком.

Теперь, когда мы **поговорили** об абстрактных принципах интерфейса виртуального компьютера, нам осталось всего ничего — реализовать все эти функции...

**СОВЕТ**

То, чем мы **только что занимались**, — очень полезный **урок** дизайна. Мы разработали **переносимую графическую систему, в которой для реального переноса на другую платформу надо разобратся с "начинкой" очень небольшого количества функций**.

## Консоль игры T3DLIB

Теперь передо мной новая дилемма. Я хочу показать вам прототип обобщенной консоли игры для Windows, которую мы будем использовать в качестве оболочки для наших **игр**. Однако в ней используются вызовы функций API. Должен ли я показать вызовы библиотечных функций сейчас или мне следует подождать? Но я подумал — ведь эта книга предназначена для более-менее опытных программистов, так что вы всегда сумеете самостоятельно обратиться к списку функций библиотеки и их описаниям, если вам что-то будет непонятно.

## Обзор T3DLIB

Теперь, когда мы обсудили конструкцию нашего виртуального компьютера, наша цель состоит в абстрагировании модели Win32/DirectX в очень простую графическую систему с двойной буферизацией и поддержкой устройств ввода и воспроизведения звука. Для этого я создал то, что громко именую *Консолью игры T3D*. Это — первый шаг на пути создания интерфейса виртуального компьютера. Перед тем как приступить к DirectX, устройствам ввода, звуку — давайте разберемся с Windows. Наилучшим способом сделать это, с моей точки зрения, является создание шаблона, или, как я его называю, консоли игры, после чего, с точки зрения программиста, приложение Windows будет выглядеть аналогично стандартному приложению DOS/UNIX.

Итак, приступим к созданию консоли игры, которая постепенно перерастет в наш виртуальный компьютер.

## Базовая консоль игры

Первое приближение консоли должно обеспечивать выполнение **следующих** задач.

1. Открытие окна.
2. Вызов **пользовательской** функции инициализации `Game_Init()`.
3. Вход в главный цикл событий Windows, обработка **сообщений** и возврат.
4. Вызов пользовательской функции `Game_Main()`, которая выполняет один цикл логики игры.

5. Циклический переход к п.3, пока пользователь не запросит завершения работы приложения.
6. Вызов пользовательской функции завершения работы и освобождения ресурсов `Game_Shutdown()`.

На рис. 3.7 приведена схема консоли игры. Обратите внимание на вызовы функций `Game_Init()`, `Game_Main()` и `Game_Shutdown()`.

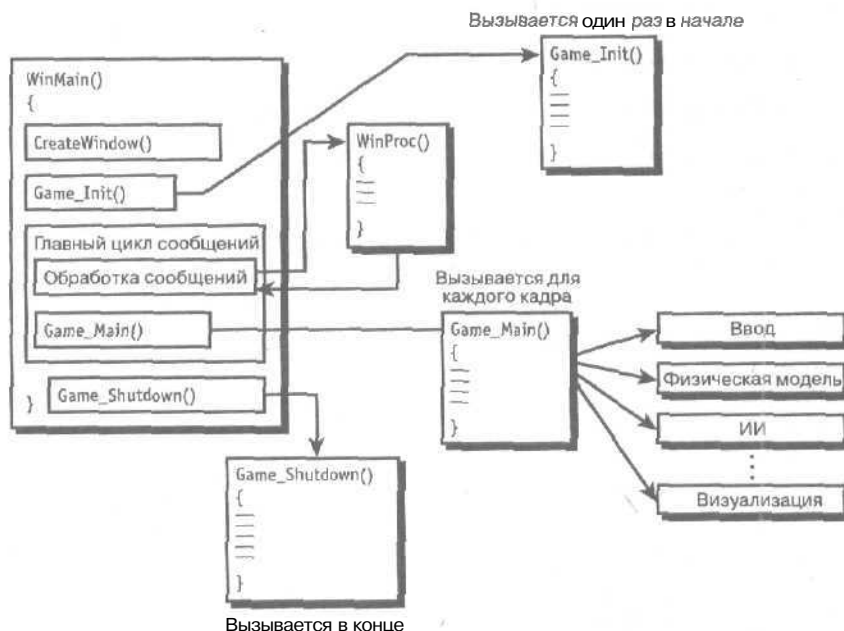


Рис. 3.7. Поток управления консоли игры

Если мы можем реализовать программу-оболочку, которая выполняет все функции, перечисленные в пп. 1–6, мы можем полностью забыть о Windows и заниматься только пользовательскими функциями `Game_*`. Начнем с прототипов этих функций.

```

// Консоль игры
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);
  
```

Не слишком сложно, не правда ли? Функции написаны так, чтобы в них можно было передавать (при необходимости) какие-то параметры (например, посредством указателя на структуры), но сам я эту возможность использовать не буду. Итак, теперь нам нужна программа, которая будет делать следующее.

1. Создавать класс Windows и регистрировать его.
2. Создавать окно.
3. Иметь цикл событий и обработчик сообщений `WinProc()`, который будет обрабатывать сообщения `WM_CREATE`, `WM_PAINT` и `WM_DESTROY`.
4. Перед вызовом цикла событий вызывать `Game_Init()` для выполнения инициализации игры.

5. В каждом цикле событий вызывать функцию `Game_Main()`, что обеспечивает работу игры.
6. По окончании работы вызывать `Game_Shutdown()` для корректного завершения игры с освобождением всех ресурсов.

Далее приведен текст файла `T3DCONSOLEALPHA1.CPP`, расположенного на прилагаемом компакт-диске. Это — завершенная программа, которая выполняет все перечисленное.

// T3DCONSOLEALPHA1.CPP - Первое приближение консоли игры

```
// INCLUDES ////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN

#include <windows.h> // Включение функциональности Windows
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // Включение функциональности C/C++
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// Макроопределения ////////////////////////////////////////

// Определения интерфейса Windows
#define WINDOW_CLASS_NAME "WIN3DCLASS" // Имя класса
#define WINDOW_TITLE "T3D Graphics Console Alpha 1.0"
#define WINDOW_WIDTH 320 // Размер окна
#define WINDOW_HEIGHT 240

// Асинхронное чтение клавиатуры
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) \
& 0x8000) ? 1 : 0)
#define KEY_UP(vk_code) ((GetAsyncKeyState(vk_code) \
& 0x8000) ? 0 : 1)

// Прототипы ////////////////////////////////////////

// Консоль игры
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// Глобальные переменные ////////////////////////////////////////

HWND main_window_handle = NULL; // Дескриптор окна
HINSTANCE main_instance=NULL; // Экземпляр приложения
char buffer[256]; // Для вывода текста
```

```

// Функции //////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    // Обработчик системных сообщений
    PAINTSTRUCT ps; // Используется в WM_PAINT
    HDC hdc; // Дескриптор контекста устройства

    // Какое это сообщение?
    switch (msg)
    {
        case WM_CREATE:
        {
            // Здесь выполняется инициализация
            return(0);
        } break;

        case WM_PAINT:
        {
            // Начало рисования
            hdc = BeginPaint(hwnd,&ps);

            // Окончание рисования
            EndPaint(hwnd,&ps);
            return(0);
        } break;

        case WM_DESTROY:
        {
            // Закрытие приложения
            PostQuitMessage(0);
            return(0);
        } break;

        default: break;

    } // switch

    // Обработка необработанных сообщений системой
    return (DefWindowProc(hwnd,msg,wparam,lparam));
} // WinProc

// WINMAIN //////////////////////////////////////

int WINAPI WinMain(HINSTANCE hinstance,
    HINSTANCE hprevinstance,
    LPSTR lpcmdline,
    int ncmdshow)
{
    WNDCLASS winclass; // Создаваемый класс
    HWND hwnd; // Дескриптор окна
    MSG msg; // Сообщение

```

```

HDC     hdc;    // Контекст устройства
PAINTSTRUCT ps;    // Структура для рисования

// Заполняем структуру класса
winclass.style = CS_DBLCLKS | CS_OWNDC |
                CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground =
    (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = WINDOW_CLASS_NAME;

// Регистрация класса окна
if (!RegisterClass(&winclass))
    return(0);

// Создание окна
if (!{hwnd=CreateWindow(
    WINDOW_CLASS_NAME, // Класс
    WINDOW_TITLE,     // Заголовок
    WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION,
    0,0,               // x,y
    WINDOW_WIDTH,      // Ширина
    WINDOW_HEIGHT,     // Высота
    NULL,              // Дескриптор родителя
    NULL,              // Дескриптор меню
    hinstance,         // Экземпляр
    NULL))) // Параметры создания
    return (0);

// Сохранение дескриптора окна и экземпляра в глобальных
// переменных
main_window_handle = hwnd;
main_instance = hinstance;

// Делаем окно видимым
ShowWindow(main_window_handle, SW_SHOW);

// Инициализация консоли игры
Game_Init();

// Вход в главный цикл событий
while(1)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // Проверка, не запрошен ли выход из программы
        if (msg.message == WM_QUIT)
            break;

        // Преобразование горячих клавиш
        TranslateMessage(&msg);
    }
}

```

```

        // Пошлaем сообщeниe процедурe Windows
        DispatchMessage(&msg);
    } // end if

    // Главная процедура игры
    Game_Main();
} // while

// Закрытие игры и освобождение ресурсов
Game_Shutdown();

// Возврат в Windows
return(msg.wParam);
} // WinMain

// Функции консоли игры //////////////////////////////////////
int Game_Init(void *parms)
{
    // В этой функции выполняется вся инициализация игры

    // Возврат кода успешного завершения
    return(1);
} // Game_Init

////////////////////////////////////
int Game_Shutdown(void *parms)
{
    // Эта функция завершает работу игры и освобождает все
    // захваченные ресурсы

    // Возврат кода успешного завершения
    return(1);
} // Game_Shutdown

////////////////////////////////////
int Game_Main(void *parms)
{
    // Эта функция вызывается в реальном времени в каждом
    // цикле событий. Вся логика игры сосредоточена в этой
    // функции

    // Проверка, не запрошен ли выход из игры
    if (KEY_DOWN(VK_ESCAPE))
    {
        PostMessage(main_window_handle, WM_DESTROY, 0, 0);
    } // if

    // Возврат кода успешного завершения
    return(1);
} // Game_Main

////////////////////////////////////

```

Имя файла с исходным кодом — `T3DCONSOLEALPHA1.CPP`; само собой разумеется, исполняемый файл имеет имя `T3DCONSOLEALPHA1.EXE`. Если вы запустите его, то увидите на экране пустое окно, показанное на рис. 3.8. Однако на самом деле сделано очень много: создано окно, вызвана функция `Game_Init()`, постоянно вызывается функция `Game_Main()`... Наконец, при закрытии окна вызывается функция `Game_Shutdown()`.

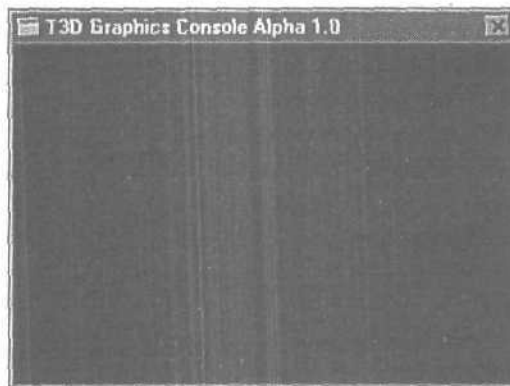


Рис. 3.8. Первое приближение консоли игры в действии

Все, что требуется от вас, — это добавить необходимую функциональность в три функции `Game_*`.

**НА ЗАМЕТКУ**

Как видите, совершенно напрасно большинство книг утверждает, что программирование для Windows — очень сложная работа. Вы уже убедились, что это просто сплошное удовольствие!

У нас уже имеется все необходимая функциональность Windows, и теперь нам необходима функциональность DirectX. Нам нужен интерфейс DirectX, который бы эмулировал наш виртуальный компьютер и предоставил в наше распоряжение графическую систему с двойной буферизацией и возможностями воспроизведения звуковых эффектов и музыки. Сейчас можно познакомиться с окончательной версией консоли игры, которая предоставляет все это в наше распоряжение, но мне представляется, что будет лучше рассмотреть API, который я использовал для создания консоли игры. Не забывайте, что нет необходимости детально разбираться с API и каждой его функцией. Просто ознакомьтесь с ними и с прилагаемыми к ним пояснениями и примерами.

После того как вы ознакомитесь с инструментарием, который будет использован для построения окончательной версии консоли игры, мы возьмем файл `T3DCONSOLEALPHA1.CPP` в качестве основы и добавим к нему различную функциональность, необходимую для получения окончательной версии интерфейса виртуального компьютера. Затем этот "шаблон" будет использоваться в качестве стартовой точки для всех демонстрационных программ и игр в данной книге. Кроме того, в конце этой главы я приведу ряд примеров, в которых покажу, как использовать графику, ввод и звуковые возможности библиотеки `T3DLIB`.

## Библиотека `T3DLIB1`

Теперь мы готовы рассмотреть все макроопределения, структуры данных и функции, составляющие API графического модуля библиотеки `T3DLIB` — `T3DLIB1.CPP`.

Модуль состоит из двух файлов — `T3DLIB1.CPP` и `H`. При работе вы просто связываете эти файлы с вашей программой и затем используете их API.

Если вы читали мою предыдущую книгу, то должны заметить, что разрабатываемый нами игровой процессор практически такой же, как и предыдущий, — с дополнительной поддержкой 16-битового режима и поддержкой окон. Код остается полностью совместимым, так что вы можете взять любой пример из первой книги и скомпилировать его с новой версией T3DLIB1.CPP без каких бы то ни было проблем. И конечно, весь код компилируется как с DirectX 8.0, так и с DirectX 9.0+.

## Архитектура графического процессора DirectX

Библиотека T3DLIB1 представляет собой двумерный игровой процессор, как видно из схемы, приведенной на рис. 3.9. Это процессор, поддерживающий работу в 8- и 16-битовом цветовых режимах, двойную буферизацию, поддержку различных разрешений экрана и отсекание. Поддерживается как оконный, так и полноэкранный режимы работы. В любом режиме работы вывод выполняется во внеэкранный буфер, содержимое которого затем копируется в первичный буфер (либо выполняется переключение буферов).

Для построения приложения с использованием библиотеки вы должны включить в проект файлы T3DLIB1.CPP\H, а также файлы DDRAW.LIB (библиотека DirectDraw) и WINMM.LIB (библиотека мультимедиа Win32).

Библиотека WINMM.LIB нужна только в том случае, когда вы используете Visual C++.

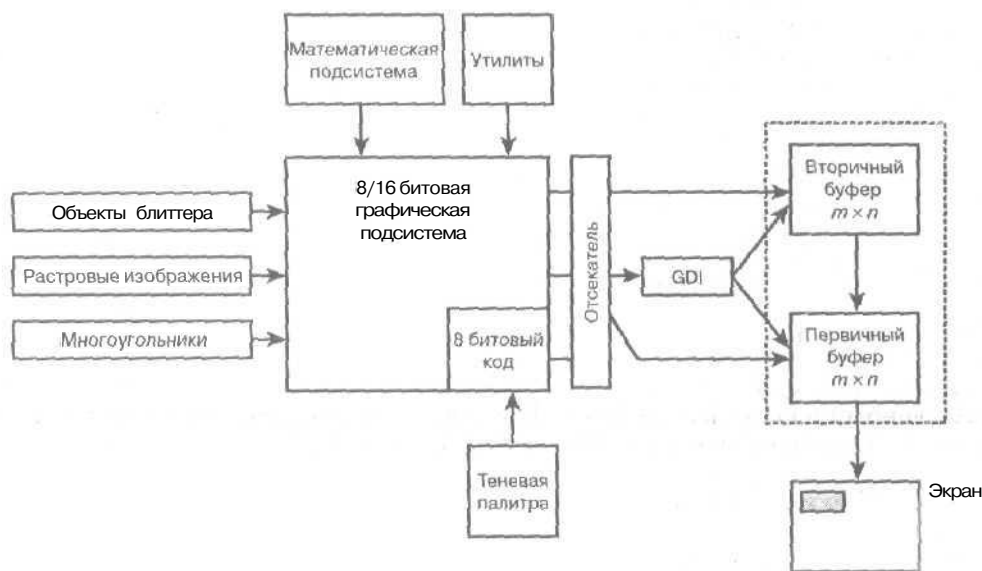


Рис. 3.9. Архитектура графического процессора

## Основные определения

Библиотека имеет один заголовочный файл — T3DLIB1.H, внутри которого имеется ряд макроопределений `#define`, используемых игровым процессором.

```
// Проверка множественного включения
#ifndef T3DLIB1
#define T3DLIB1
```

```
// DEFINES //////////////////////////////////////
```

```
// Значения по умолчанию для экрана. Могут быть
```

```
// переопределены функцией DDraw_Init()
```

```
#define SCREEN_WIDTH 640 // Размер экрана
```

```
#define SCREEN_HEIGHT 480
```

```
<define SCREEN_BPP 8 // Битов на пиксель
```

```
#define MAX_COLORS_PALETTE 256
```

```
#define DEFAULT_PALETTE_FILE "PALDATA2.PAL"
```

```
// Используется для выбора режима экрана (полноэкранный или  
// оконный)
```

```
#define SCREEN_FULLSCREEN 0
```

```
#define SCREEN_WINDOWED 1
```

```
// Определения для работы с растровыми изображениями
```

```
#define BITMAP_ID 0x4D42 // Универсальный идентификатор
```

```
// растрового изображения
```

```
#define BITMAP_STATE_DEAD 0
```

```
#define BITMAP_STATE_ALIVE 1
```

```
#define BITMAP_STATE_DYING 2
```

```
#define BITMAP_ATTR_LOADED 128
```

```
#define BITMAP_EXTRACT_MODE_CELL 0
```

```
#define BITMAP_EXTRACT_MODE_ABS 1
```

```
// Форматы пикселей DirectDraw
```

```
#define DD_PIXEL_FORMAT8 8
```

```
#define DD_PIXEL_FORMAT555 15
```

```
#define DD_PIXEL_FORMAT565 16
```

```
#define DD_PIXEL_FORMAT888 24
```

```
#define DD_PIXEL_FORMATALPHA888 32
```

```
// Определения для объектов блиттера
```

```
<define BOB_STATE_DEAD 0
```

```
#define BOB_STATE_ALIVE 1
```

```
#define BOB_STATE_DYING 2
```

```
#define BOB_STATE_ANIM_DONE 1
```

```
<define MAX_BOB_FRAMES 64
```

```
#define MAX_BOB_ANIMATIONS 16
```

```
#define BOB_ATTR_SINGLE_FRAME 1
```

```
#define BOB_ATTR_MULTI_FRAME 2
```

```
#define BOB_ATTR_MULTI_ANIM 4
```

```
#define BOB_ATTR_ANIM_ONE_SHOT 8
```

```
<define BOB_ATTR_VISIBLE 16
```

```
#define BOB_ATTR_BOUNCE 32
```

```
#define BOB_ATTR_WRAPAROUND 64
```

```
#define BOB_ATTR_LOADED 128
```

```
<define BOB_ATTR_CLONE 256
```

```
// Команды перехода экрана (режим 256 цветов)
```

```
#define SCREEN_DARKNESS 0 // Переход к черному
```

```

#define SCREEN_WHITENESS 1 // Переход к белому
#define SCREEN_SWIPE_X 2 // Горизонтальное стирание
#define SCREEN_SWIPE_Y 3 // Вертикальное стирание
#define SCREEN_DISOLVE 4 // "Растворение"
#define SCREEN_SCRUNCH 5 // Квадратичная компрессия
#define SCREEN_BLUENESS 6 // П... к синему
#define SCREEN_REDNESS 7 // Переход к красному
#define SCREEN_GREENNESS 8 // Переход к зеленому

// Определения для мигающих цветов
#define BLINKER_ADD 0 // Добавка света в базу данных
#define BLINKER_DELETE 1 // Удаление цвета из базы данных
#define BLINKER_UPDATE 2 // Обновление цвета
#define BLINKER_RUN 3 // Нормальная работа

// Определения PI
#define PI ((float)3.141592654f)
#define PI2 ((float)6.283185307f)
#define PI_DIV_2 ((float)1.570796327f)
#define PI_DIV_4 ((float)0.785398163f)
#define PI_INV ((float)0.318309886f)

// Константы для работы с числами в формате с фиксированной
// точкой
#define FIXP16_SHIFT 16
#define FIXP16_MAG 65536
#define FIXP16_DP_MASK 0x0000ffff
#define FIXP16_WP_MASK 0xffff0000
#define FIXP16_ROUND_UP 0x00008000

```

## Макросы

Перечисленные здесь макросы будут встречаться в самых разных местах; здесь же они собраны вместе.

```

// Асинхронное чтение клавиатуры
#define KEY_DOWN(vk_code) \
    ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code) \
    ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// Построение 16-битового цвета в формате 5.5.5
#define _RGB16BIT555(r,g,b) \
    ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))

// Построение 16-битового цвета в формате 5.6.5
#define _RGB16BIT565(r,g,b) \
    ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))

// Построение 24-битового цвета в формате 8.8.8
#define _RGB24BIT(a,r,g,b) \
    ((b) + ((g) << 8) + ((r) << 16))

```

```

// Построение 32-битового цвета в формате A.8.8.8
#define _RGB32BIT(a,r,g,b) \
    ((b) + ((a) << 8) + ((r) << 16) + ((a) << 24))

// Макросы для работы с битами
#define SET_BIT(word,bit_flag) \
    ((word) = ((word) | (bit_flag)))
#define RESET_BIT(word,bit_flag) \
    ((word) = ((word) & ~(bit_flag)))

// Инициализация структуры DirectDraw
// нулями и указание поля dwSize
#define DDRAW_INIT_STRUCT(ddstruct) \
    { memset(&ddstruct,0,sizeof(ddstruct)); \
      ddstruct.dwSize = sizeof(ddstruct); }

// Вычисление минимума и максимума двух выражений
#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAX(a,b) (((a) > (b)) ? (a) : (b))

// Обмен значений
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// Некоторые математические макросы
#define DEG_TO_RAD(ang) ((ang)*PI/180)
#define RAD_TO_DEG(rads) ((rads)*180/PI)
#define RAND_RANGE(x,y) ((x) + (rand()%(y)-(x)+1))

```

## Типы и структуры данных

Теперь мы ознакомимся с типами и структурами данных, используемыми игровым процессором.

```

// Основные беззнаковые типы
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char UCHAR;
typedef unsigned char BYTE;
typedef unsigned int QUAD;
typedef unsigned int UINT;

// Структура для хранения .BMP-файлов с растровыми
// изображениями
typedef struct BITMAP_FILE_TAG
{
    // Заголовок файла с растровым изображением
    BITMAPFILEHEADER bitmapfileheader;
    // Информация, включающая палитру
    BITMAPINFOHEADER bitmapinfoheader;
    PALETTEENTRY palette[256]; // Палитра
    UCHAR *buffer; // Указатель на данные
} BITMAP_FILE, *BITMAP_FILE_PTR;

```

```

// Структура объекта блиттера
typedef struct BOB_TYP
{
    int state;        // Состояние объекта
    int anim_state;   // Состояние анимации
    int attr;         // Атрибуты объекта
    float x,y;        // Позиция вывода изображения
    float xv,yv;      // Скорость объекта
    int width, height; // Ширина и высота
    int width_fill;   // Внутренняя переменная
    int bpp;          // Битов на пиксель
    int counter_1;    // Общие счетчики
    int counter_2;
    int max_count_1;  // Пороговые значения
    int max_count_2;
    int varsI[16];    // Стек из 16 целых чисел
    float varsF[16];  // Стек из 16 действительных чисел
    int curr_frame;   // Текущий кадр анимации
    int num_frames;   // Общее количество кадров анимации
    int curr_animation; // Индекс текущей анимации
    int anim_counter; // Используется при анимации
    int anim_index;   // Индекс элемента анимации
    int anim_count_max; // Количество циклов до анимации
    // Последовательности анимации
    int *animations[MAX_BOB_ANIMATIONS];
    // Поверхности
    LPDIRECTDRAWSURFACE7 images[MAX_BOB_FRAMES];
} BOB, *BOB_PTR;

// Простое растровое изображение
typedef struct BITMAP_IMAGE_TYP
{
    int state;        // Состояние изображения
    int attr;         // Атрибуты изображения
    int x,y;          // Позиция изображения
    int width, height; // Размер изображения
    int num_bytes;    // Количество байтов изображения
    int bpp;          // Битов на пиксель
    UCHAR *buffer;    // Пиксели изображения
} BITMAP_IMAGE, *BITMAP_IMAGE_PTR;

// Структуры мигающего света
typedef struct BLINKER_TYP
{
    int color_index;   // Индекс цвета
    PALETTEENTRY on_color; // Цвет включенного состояния
    PALETTEENTRY off_color; // Цвет выключенного состояния
    int on_time;       // Количество кадров во
                        // включенном состоянии
    int off_time;      // Количество кадров в
                        // выключенном состоянии
    // Внутренние члены
    int counter;       // Счетчик переходов состояний
}

```

```

    int state;          // Состояние света:
                        // -1 - выключен, 1 - включен,
                        // 0 - отключен
> BLINKER, *BLINKER_PTR;

// Двумерная вершина
typedef struct VERTEX2DI_TYP
{
    int x,y;           // Вершина
} VERTEX2DI, *VERTEX2DI_PTR;

// Двумерная вершина
typedef struct VERTEX2DF_TYP
{
    float x,y;         // Вершина
} VERTEX2DF, *VERTEX2DF_PTR;

// Двумерный многоугольник
typedef struct POLYGON2D_TYP
{
    int state;          // Состояние многоугольника
    int num_verts;      // Число вершин
    int x0,y0;          // Позиция центра
    int xv,yv;          // Начальная скорость
    DWORD color;         // Индекс или PALETTEENTRY
    VERTEX2DF *Mist;     // Указатель на список вершин
} POLYGON2D, *POLYGON2D_PTR;

// Определения матриц
typedef struct MATRIX3X3_TYP
{
    union
    {
        float M[3][3]; // Массив индексированных данных для
                        // хранения матрицы в формате со
                        // старшей строкой
        struct
        {
            float M00, M01, M02;
            float M10, M11, M12;
            float M20, M21, M22;
        };
    }; // union
} MATRIX3X3, *MATRIX3X3_PTR;

typedef struct MATRIX1X3_TYP
{
    union
    {
        float M[3]; // Массив индексированных данных для
                    // хранения матрицы
        struct
        {

```

```

        float M00, M01, M02;
    >
}; // union
} MATRIX1X3, *MATRIX1X3_PTR;

typedef struct MATRIX3X2_TYP
{
    union
    {
        float M[3][2]; // Массив индексированных данных для
                        // хранения матрицы в формате со
                        // старшей строкой

        struct
        {
            float M00, M01;
            float M10, M11;
            float M20, M21;
        };
    }; // union
} MATRIX3X2, *MATRIX3X2_PTR;

typedef struct MATRIX1X2_TYP
(
    union
    {
        float M[2]; // Массив индексированных данных для
                    // хранения матрицы

        struct
        {
            float M00, M01;
        };
    }; // union
} MATRIX1X2, *MATRIX1X2_PTR;

```

Обратите внимание на некоторую поддержку математических операций. Она связана с поддержкой двумерных преобразований многоугольников в предыдущей книге.

## Прототипы функций

Теперь, перед тем как приступить к рассмотрению отдельных функций, я хочу познакомить вас с их полным списком.

```

// Функции DirectDraw
int DDraw_Init(int width, int height, int bpp,
               int windowed=0);
int DDraw_Shutdown(void);

LPDIRECTDRAWCLIPPER
DDraw_Attach_Clipper(LPDIRECTDRAWSURFACE7 lpdds,
                    int num_rects, LPRECT clip_list);

LPDIRECTDRAWSURFACE7
DDraw_Create_Surface(int width, int height,
                    int mem_flags=0,

```

```

        USHORT color_key_value=0);

int DDraw_Flip(void);

int DDraw_Wait_For_Vsync(void);

int DDraw_Fill_Surface(LPDIRECTDRAW SURFACE7 lpdds,
        USHORT color, RECT *client=NULL);

UCHAR *DDraw_Lock_Surface(LPDIRECTDRAW SURFACE7 lpdds,
        int *lpitch);

int DDraw_Unlock_Surface(LPDIRECTDRAW SURFACE7 lpdds);
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);
UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);

// Функции для работы с объектами блиттера
int Create_BOB(BOB_PTR bob,int x, int y,int width,
        int height, int num_frames,int attr,
        int mem_flags=0, USHORT color_key_value=0,
        int bpp=8);

int Clone_BOB(BOB_PTR source, BOB_PTR dest);
int Destroy_BOB(BOB_PTR bob);
int Draw_BOB(BOB_PTR bob, LPDIRECTDRAW SURFACE7 dest);

int Draw_Scaled_BOB(BOB_PTR bob, int swidth, int sheight
        LPDIRECTDRAW SURFACE7 dest);

int Draw_BOB16(BOB_PTR bob, LPDIRECTDRAW SURFACE7 dest);

int Draw_Scaled_BOB16(BOB_PTR bob, int swidth, int sheight
        LPDIRECTDRAW SURFACE7 dest);

int Load_Frame_BOB(BOB_PTR bob, BITMAP_FILE_PTR bitmap,
        int frame, int cx,int cy,int mode);

int Load_Frame_BOB16(BOB_PTR bob, BITMAP_FILE_PTR bitmap,
        int frame, int cx,int cy,int mode);

int Animate_BOB(BOB_PTR bob);
int Move_BOB(BOB_PTR bob);

int Load_Animation_BOB(BOB_PTR bob, int anim_index,
        int num_frames, int *sequence);

int Set_Pos_BOB(BOB_PTR bob, int x, int y);
int Set_Vel_BOB(BOB_PTR bob,int xv, int yv);
int Set_Anim_Speed_BOB(BOB_PTR bob,int speed);
int Set_Animation_BOB(BOB_PTR bob, int anim_index);
int Hide_BOB(BOB_PTR bob);

```

```

int Show_BOB(BOB_PTR bob);
int Collision_BOBS(BOB_PTR bob1, BOB_PTR bob2);

// Вспомогательные функции общего назначения
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);

int Collision_Test(int x1, int y1, int w1, int h1,
                  int x2, int y2, int w2, int h2);

int Color_Scan(int x1, int y1, int x2, int y2,
               UCHAR scan_start, UCHAR scan_end,
               UCHAR *scan_buffer, int scan_lpitch);

int Color_Scan16(int x1, int y1, int x2, int y2,
                 USHORT scan_start, USHORT scan_end,
                 UCHAR *scan_buffer, int scan_lpitch);

// Графические функции
int Draw_Clip_Line(int x0, int y0, int x1, int y1, int color,
                  UCHAR *dest_buffer, int lpitch);

int Draw_Clip_Line16(int x0, int y0, int x1, int y1,
                    int color, UCHAR *dest_buffer,
                    int lpitch);

int Clip_Line(int &x1, int &y1, int &x2, int &y2);

int Draw_Line(int x0, int y0, int x1, int y1, int color,
              UCHAR *vb_start, int lpitch);

int Draw_Line16(int x0, int y0, int x1, int y1, int color,
                UCHAR *vb_start, int lpitch);

int Draw_Pixel(int x, int y, int color,
               UCHAR *video_buffer, int lpitch);

int Draw_Rectangle(int x1, int y1, int x2, int y2,
                   int color, LPDIRECTDRAWSURFACE7 lpdds);

void HLine(int x1, int x2, int y, int color,
           UCHAR *vbuffer, int lpitch);

void VLine(int y1, int y2, int x, int color,
           UCHAR *vbuffer, int lpitch);

void HLine16(int x1, int x2, int y, int color,
             UCHAR *vbuffer, int lpitch);

void VLine16(int y1, int y2, int x, int color,
             UCHAR *vbuffer, int lpitch);

```

```

void Screen_Transitions(int effect UCHAR *vbuffer,
                        int lpitch);

int Draw_Pixel(int x, int y, int color,
              UCHAR *video_buffer, int lpitch);

int Draw_Pixel16(int x, int y, int color,
                UCHAR *video_buffer, int lpitch);

// Функции для работы с палитрой
int Set_Palette_Entry(int color_index,
                    LPPALETTEENTRY color);

int Get_Palette_Entry(int color_index,
                    LPPALETTEENTRY color);

int Load_Palette_From_File(char *filename,
                          LPPALETTEENTRY palette);

int Save_Palette_To_File(char *filename,
                        LPPALETTEENTRY palette);

int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
int Rotate_Colors(int start_index, int end_index);

int Blink_Colors(int command, BLINKER_PTR new_light,
                int id);

// Функции для работы с растровыми изображениями
int Create_Bitmap(BITMAP_IMAGE_PTR image, int x, int y,
                 int width, int height, int bpp=8);

int Destroy_Bitmap(BITMAP_IMAGE_PTR image);

int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap,
               UCHAR *dest_buffer,
               int lpitch, int transparent);

int Draw_Bitmap16(BITMAP_IMAGE_PTR source_bitmap,
                 UCHAR *dest_buffer,
                 int lpitch, int transparent);

int Load_Image_Bitmap(BITMAP_IMAGE_PTR image,
                     BITMAP_FILE_PTR bitmap,
                     int cx, int cy, int mode);

int Load_Image_Bitmap16(BITMAP_IMAGE_PTR image,
                       BITMAP_FILE_PTR bitmap,
                       int cx, int cy, int mode);

int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);

```

```

int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap, int dest_x,
int dest_y, BITMAP_IMAGE_PTR source_bitmap,
int source_x, int source_y, int width,
int height);

int Flip_Bitmap(UCHAR *image, int bytes_per_line,
int height);

// Функции для работы с файлами растровых изображений
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap,
char *filename);

int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap);

// Функции GDI
int Draw_Text_GDI(char *text, int x, int y, COLORREF color,
LPDIRECTDRAWSURFACE7 lpdds);

int Draw_Text_GDI(char *text, int x, int y, int color,
LPDIRECTDRAWSURFACE7 lpdds);

// Функции обработки ошибок
int Open_Error_File(char *filename, FILE *fp_override=NULL);
int Close_Error_File(void);
int Write_Error(char *string, ...);

// Двумерная визуализация треугольников
void Draw_Top_Tri(int x1, int y1, int x2, int y2, int x3,
int y3, int color,
UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_Tri(int x1, int y1, int x2, int y2,
int x3, int y3, int color,
UCHAR *dest_buffer, int mempitch);

void Draw_Top_Tri16(int x1, int y1, int x2, int y2, int x3,
int y3, int color, UCHAR *dest_buffer,
int mempitch);

void Draw_Bottom_Tri16(int x1, int y1, int x2, int y2,
int x3, int y3, int color,
UCHAR *dest_buffer, int mempitch);

void Draw_Top_TriFP(int x1, int y1, int x2, int y2, int x3,
int y3, int color, UCHAR *dest_buffer,
int mempitch);

void Draw_Bottom_TriFP(int x1, int y1, int x2, int y2,
int x3, int y3, int color,
UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D(int x1, int y1, int x2, int y2,
int x3, int y3, int color,

```

```

        UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D16(int x1, int y1, int x2, int y2,
    int x3, int y3, int color,
    UCHAR *dest_buffer, int mempitch);

void Draw_TriangleFP_2D(int x1, int y1, int x2, int y2,
    int x3, int y3, int color,
    UCHAR *dest_buffer, int mempitch);

inline void Draw_QuadFP_2D(int x0, int y0, int x1, int y1,
    int x2, int y2, int x3, int y3,
    int color, UCHAR *dest_buffer,
    int mempitch);

// Двумерная визуализация и преобразование многоугольников
void Draw_Filled_Polygon2D(POLYGON2D_PTR poly,
    UCHAR *vbuffer, int mempitch);

void Draw_Filled_Polygon2D16(POLYGON2D_PTR poly,
    UCHAR *vbuffer, int mempitch);

int Translate_Polygon2D(POLYGON2D_PTR poly, int dx, int dy);
int Rotate_Polygon2D(POLYGON2D_PTR poly, int theta);
int Scale_Polygon2D(POLYGON2D_PTR poly, float sx, float sy);
void Build_Sin_Cos_Tables(void);

int Translate_Polygon2D_Mat(POLYGON2D_PTR poly,
    int dx, int dy);

int Rotate_Polygon2D_Mat(POLYGON2D_PTR poly, int theta);

int Scale_Polygon2D_Mat(POLYGON2D_PTR poly,
    float sx, float sy);

int Draw_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer,
    int lpitch);

int Draw_Polygon2D16(POLYGON2D_PTR poly, UCHAR *vbuffer,
    int lpitch);

// Математические функции
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

int Find_Bounding_Box_Poly2D(POLYGON2D_PTR poly,
    float &min_x, float &max_x,
    float &min_y, float &max_y);

int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
    MATRIX3X2_PTR mb,
    MATRIX1X2_PTR mprod);

```

```

int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma,
                    MATRIX3X3_PTR mb,
                    MATRIX1X3_PTR mprod);

int Mat_Mul_3X3(MATRIX3X3_PTR ma,
                MATRIX3X3_PTR mb,
                MATRIX3X3_PTR mprod);

inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
                        float m00, float m01,
                        float m10, float m11,
                        float m20, float m21);

// Функции для работы с памятью
inline void Mem_Set_WORD(void *dest, USHORT data,
                          int count);
inline void Mem_Set_QUAD(void *dest, UINT data,
                          int count);

```

Обратите внимание на то, что все вызовы функций растеризации получают указатель на буфер кадра типа `UCHAR*`. Кроме того, все шаги памяти также имеют значения, выраженные в байтах (вспомните — мы договаривались об этом ранее). Заметьте также, несмотря на то, что 90% функций ориентировано на 16-битовую графику, некоторые функции работают только с 8-битовым цветом. И наконец, большое количество функций для работы с многоугольниками в двумерном пространстве объясняется тем, что они оказались унаследованы из первой книги.

## Глобальные переменные

Вы знаете о моем равнодушном отношении к глобальным переменным из-за скорости работы с ними. Кроме того, они вполне подходят для использования на уровне системы. Вот список глобальных переменных, используемых нашей библиотекой.

```

FILE *fp_error; // Файл для сообщения об ошибках
char error_filename[80]; // Имя файла ошибок

// Обратите внимание на использование интерфейса версии 7.0
LPDIRECTDRAW7 lpdd; // Объект dd
LPDIRECTDRAW_SURFACE7 lpddsprimary; // Первичная поверхность dd
LPDIRECTDRAW_SURFACE7 lpddsback; // Вторичная поверхность dd
LPDIRECTDRAW_PALETTE lpddpal; // Указатель на созданную
// палитру dd
LPDIRECTDRAW_CLIPPER lpddclipper; // Отсекатель для
// вторичной поверхности dd
LPDIRECTDRAW_CLIPPER lpddclipperwin; // Отсекатель для окна
PALETTEENTRY palette[256]; // Палитра
PALETTEENTRY save_palette[256]; // Используется для
// сохранения палитры
DDSURFACEDESC2 ddsd; // Структура описания поверхности
// DirectDraw
DDBLTFX ddbltfx; // Используется для заполнения
DDSCAPS2 ddscaps; // Структура описания возможностей
// поверхности

```

```

HRESULT ddrval; // Результат вызовов dd
UCHAR *primary_buffer; // Первичный видеобuffer
UCHAR *back_buffer; // Вторичный видеобuffer
int primary_pitch; // Шаг памяти
int back_pitch; // Шаг памяти
BITMAP_FILE bitmap8bit; // 8-битовый растровый файл
BITMAP_FILE bitmap16bit; // 16-битовый растровый файл
BITMAP_FILE bitmap24bit; // 24-битовый растровый файл

DWORD start_clock_count; // Используется при синхронизации
int windowed_mode; // Режим работы - оконный или
// полноэкранный

// Отсекающий прямоугольник для программного отсека
int min_clip_x, max_clip_x,
min_clip_y, max_clip_y;

// Эти переменные переопределяются вызовом DD_Init()
int screen_width, // Ширина экрана
screen_height, // Высота экрана
screen_bpp, // Битов на пиксель
screen_windowed; // Режим работы

int dd_pixel_format; // Формат пикселей по умолчанию
int window_client_x0; // Координаты клиентской области
int window_client_y0; // в оконном режиме

// Таблицы поиска для быстрых вычислений
float cos_look[360];
float sin_look[360];

// Указатель функции для построения данных в формате 5.5.5
// или 5.6.5 — в зависимости от используемого режима
USHORT (*RGB16Bit)(int r, int g, int b);

```

## Интерфейс DirectDraw

Теперь мы можем приступить к детальному рассмотрению функций поддержки DirectDraw. Подсистема DirectDraw из первой книги обладает следующими возможностями:

- двойная буферизация первичным и вторичным (внеэкранным) буферами кадров;
- поддержка 8-битового цвета с использованием палитр;
- поддержка 16-битового цвета с автоматическим определением формата пикселей;
- поддержка оконного режима для 8- и 16-битовых цветов;
- двумерное отсечение многоугольников и растровых изображений;
- доступ к первичному и вторичному буферам;
- переключение страниц и копирование буферов для оконных приложений.

Обратите внимание на последнюю возможность. DirectX не позволяет оконному приложению обладать двойной буферизацией, поэтому такое приложение имеет только первичный буфер и программно создается вторичный буфер, размер которого соответствует размеру клиентской области окна. Затем при запросе переключения содержимое внеэкрannого буфера копируется в клиентскую область окна. И конечно же, все это происходит полностью прозрачно для программиста.

А теперь рассмотрим каждую функцию по отдельности.

#### Прототип функции

```
int DDraw_Init(int width,    // Ширина экрана
               int height   // Высота экрана
               int bpp,      // Битов на пиксель
               int windowed=0); // 0 - полноэкранный режим,
                               // 1 - оконный
```

#### Назначение

Функция `DDraw_Init()` запускает и инициализирует DirectDraw. Вы можете указать любые разрешение и глубину цвета. Если вы планируете работать в оконном режиме, передайте в качестве последнего параметра 1. При успешном завершении возвращает TRUE.

#### Пример использования

```
// Полноэкранный режим 800x600 с 256 цветами
DDraw_Init(800,600,8);
//Оконный режим 400x400 с 16-битовым цветом. Примечание:
// рабочий стол должен находиться в режиме с 16-битовым
// цветом
DDraw_Init(400,400,16,1);
```

#### Прототип функции

```
int DDraw_Shutdown(void);
```

#### Назначение

Функция `DDraw_Shutdown()` завершает работу DirectDraw и освобождает все интерфейсы.

#### Пример использования

```
// В коде завершения работы приложения
DDraw_Shutdown();
```

#### Прототип функции

```
LPDIRECTDRAWCLIPPER DDraw_Attach_Clipper(
    LPDIRECTDRAWSURFACE7 lpdds, // Поверхность, к которой
                                // присоединяется отсекатель
    int num_rects,               // Количество прямоугольников
    LPRECT clip_list);           // Указатель на
                                // прямоугольники
```

#### Назначение

Функция `DDraw_Attach_Clipper()` присоединяет отсекатель к переданной в качестве параметра поверхности (в большинстве случаев — вторичный буфер). Кроме того передает количество прямоугольников в списке отсечения и сам список. При успешном завершении возвращает TRUE.

#### Пример использования

```
// Создание отсекающей области с размером экрана
RECT clip_zone = {0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1};
DDraw_Attach_Clipper(lpddsback, 1, &clip_zone);
```

### Прототип функции

```
LPDIRECTDRAW_SURFACE7 DDraw_Create_Surface(  
    int width,           // Ширина поверхности  
    int height,          // Высота поверхности  
    int mem_flags=0,     // Управляющие флаги  
    USHORT color_key_value=0); // Значение цветового ключа
```

### Назначение

Функция `DDraw_Create_Surface()` используется для создания внеэкранной поверхности `DirectDraw` в системной памяти, видеопамяти или памяти **AGP**. По умолчанию используется флаг `DDSCAPS_OFFSCREENPLAIN`. Все прочие управляющие флаги добавляются при помощи операции побитового ИЛИ. Для указания создания поверхности в системной памяти или видеопамяти используются флаги `DDSCAPS_SYSTEMMEMORY` и `DDSCAPS_VIDEOMEMORY` соответственно. Функция обладает внутренней логикой для определения того, создание какой поверхности требуется — 8- или 16-битовой. Последний параметр определяет значение цветового ключа, которое по умолчанию равно 0. В 8-битовом режиме это значение соответствует нулевому индексу, в 16-битовом — `RGB(0,0,0)`. Если вы хотите указать другое значение ключа прозрачного цвета, передайте соответствующее значение в функцию вместо используемого по умолчанию. При успешном завершении функция возвращает указатель на новую поверхность, при неуспешном — значение `NULL`.

### Пример использования

```
// Создание поверхности 64x64 в видеопамяти  
LPDIRECTDRAW_SURFACE7 image =  
    DDraw_Create_Surface(64,64,DDSCAPS_VIDEOMEMORY);  
// Создание внеэкранной поверхности 128x128 с цветовым  
// ключом 16 (предполагается работа в режиме 256 цветов)  
LPDIRECTDRAW_SURFACE7 image =  
    DDraw_Create_Surface(128,128,DDSCAPS_OFFSCREENPLAIN,16);
```

### Прототип функции

```
int DDraw_Flip(void);
```

### Назначение

Функция `DDraw_Flip()` переключает первичную и вторичную поверхности в полноэкранном режиме; в оконном режиме она копирует вторичный буфер в клиентскую область. Возврат выполняется после реального осуществления переключения, так что он может не быть немедленным. При успешном выполнении возвращает значение `TRUE`.

### Пример использования

```
// Переключение поверхностей  
DDraw_Flip();
```

### Прототип функции

```
int DDraw_Wait_For_Vsync(void);
```

### Назначение

Функция `DDraw_Wait_For_Vsync()` ожидает обратного хода луча вертикальной развертки. Возвращает `TRUE` при успешном завершении и `FALSE`, если произошло что-то ужасное...

### Пример использования

```
// Ожидание обратного вертикального хода луча  
DDraw_Wait_For_Vsync();
```

### Прототип функции

```
int DDraw_Fill_Surface(  
LPDIRECTDRAW_SURFACE7 lpdds, // Заполняемая поверхность  
int color, // Цвет заполнения  
RECT *client = NULL); // Клиентская область либо  
// NULL для всей поверхности
```

### Назначение

Функция `DDraw_Fill_Surface()` используется для заполнения поверхности определенным цветом. Цвет должен быть указан в формате, соответствующем используемой глубине цвета, — один байт в случае 256 цветов или RGB-значение в 16-битовом режиме. При необходимости частичного заполнения вы можете указать конкретную область; в противном случае будет заполнена вся поверхность. При успешном завершении возвращает `TRUE`.

### Пример использования

```
// Заполнение первичной поверхности цветом 0  
DDraw_Fill_Surface(lpddsprimary, 0);
```

### Прототип функции

```
UCHAR *DDraw_Lock_Surface(LPDIRECTDRAW_SURFACE7 lpdds,  
int *lpitch);
```

### Назначение

Функция `DDraw_Lock_Surface()` блокирует передаваемую в качестве параметра поверхность (если это возможно) и возвращает указатель типа `UCHAR*` на данную поверхность, одновременно обновляя переменную `lpitch` значением шага памяти. Пока поверхность заблокирована, вы можете работать с ней, записывая в нее пиксели, однако **блиттинг** при этом невозможен — таким образом вы должны разблокировать поверхность, как только это предоставляется возможным. После разблокирования поверхности указатель памяти становится недействительным и не должен использоваться. При неуспешном завершении возвращает значение `NULL`.

### Пример использования

```
// Переменная для хранения шага памяти  
int lpitch = 0;  
// Блокируем поверхность  
UCHAR *memory = DDraw_Lock_Surface(image, &lpitch);
```

### Прототип функции

```
int DDraw_Unlock_Surface(LPDIRECTDRAW_SURFACE7 lpdds);
```

### Назначение

Функция `DDraw_Unlock_Surface()` разблокирует предварительно заблокированную с помощью функции `DDraw_Lock_Surface()` поверхность. В функцию передается адрес блокируемой поверхности. При успешном выполнении возвращает `TRUE`.

### Пример использования

```
// Разблокирование поверхности  
DDraw_Unlock_Surface(image);
```

### Прототипы функций

```
UCHAR *DDraw_Lock_Back_Surface(void);  
UCHAR *DDraw_Lock_Primary_Surface(void);
```

### Назначение

Эти две функции используются для блокировки первичной и вторичной поверхностей визуализации. В большинстве случаев используется только блокировка вторичной поверхности, но имеется возможность при необходимости блокировки первичной поверхности. При вызове функции `DDraw_Lock_Primary_Surface()` следующие глобальные переменные становятся корректными.

```
extern UCHAR *primary_buffer; // Первичный видеобuffer
extern int primary_lpitch; // Шаг памяти
```

После этого вы можете работать с памятью поверхности. Однако **блиттинг** при этом оказывается невозможным. Вызов `DDraw_Lock_Back_Surface()` блокирует вторичную поверхность и делает корректными следующие глобальные переменные.

```
extern UCHAR *back_buffer; // Вторичный buffer
extern int back_lpitch; // Шаг памяти
```

#### НА ЗАМЕТКУ

Не пытайтесь самостоятельно изменять значения указанных глобальных переменных — это может привести к непредсказуемому поведению игрового процессора.

### Пример использования

```
// Блокируем первичную поверхность и записываем пиксель в
// верхний левый угол (считаем, что используется 8-битовый
// режим)
DDraw_Lock_Primary();
primary_buffer[0] = 100;
```

### Прототипы функций

```
int DDraw_Unlock_Primary_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

### Назначение

Эти функции разблокируют первичную и вторичную поверхности. При разблокировании незаблокированной поверхности функция не выполняет никаких действий. При успешном завершении возвращает значение `TRUE`.

### Пример использования

```
// Разблокируем вторичную поверхность
DDraw_Unlock_Back_Surface();
```

## Функции для работы с двумерными многоугольниками

Следующий набор функций представляет собой систему для работы с двумерными многоугольниками. Ничего особо потрясающего воображение в этих функциях нет, так что нет и никакой демонстрационной программы, показывающей всю их мощь.

### Прототипы функций

```
void Draw_Triangle_2D(int x1, int y1, // Вершины треугольника
    int x2, int y2,
    int x3, int y3,
    int color, // 8-битовый цвет
    UCHAR *dest_buffer, // Buffer
    int mempitch); // Шаг памяти

// Версия с фиксированной точкой немного менее точна
```

```

void Draw_TriangleFP_2D(int x1,int y1,
    int x2,int y2,
    int x3,int y3,
    int color,
    UCHAR *dest_buffer,
    int mempitch);

// 16-битовая версия
void Draw_Triangle_2D16(int x1,int y1, // Вершины треугольника
    int x2,int y2,
    int x3,int y3,
    int color, // 16-битовый RGB-цвет
    UCHAR *dest_buffer, // Буфер
    int mempitch); // Шаг памяти

```

#### Назначение

Функции `Draw_Triangle_2D*()` чертят заполненные треугольники в указанном буфере цветом, переданным в качестве параметра. Треугольник будет отсечен текущей областью отсечения, определенной глобальными переменными (а не отсекателем `DirectDraw`). Функция `Draw_TriangleFP_2D()` выполняет те же действия, но с использованием математики с фиксированной точкой — в результате она получается немного более быстрой, но менее точной. Функции ничего не возвращают.

#### Пример использования

```

// Черчение треугольника (100,10) (150,50) (50,60)
// с цветом с индексом 50 во вторичном буфере
Draw_Triangle_2D(100,10,150,50,50,60,
    50, // Индекс цвета
    back_buffer,
    back_pitch);

// То же, но в 16-битовом режиме красным цветом
Draw_Triangle_2D16(100,10,150,50,50,60,
    RGB16BIT565(31,0,0), // Считаем, что
    back_buffer, // используется
    back_pitch); // формат 5.6.5

```

#### Прототип функции

```

inline void
Draw_QuadFP_2D(int x0, int y0, // Вершины
    int x1, int y1,
    int x2, int y2,
    int x3, int y3,
    int color, // 8-битовый индекс цвета
    UCHAR *dest_buffer, // Videобуфер
    int mempitch); // Шаг памяти

```

#### Назначение

Функция `Draw_QuadFP_2D()` чертит четырехугольник как композицию двух треугольников. Имеется только 8-битовая версия. Функция не возвращает ничего.

#### Пример использования

```

// Чертим четырехугольник. Обратите внимание на то, что
// вершины должны быть упорядочены — по часовой стрелке или
// против нее

```

```
Draw_QuadFP_2D(0,0, 10,0, 15,20,5,25,
               100, back_buffer, back_lpitch);
```

#### Прототипы функций

```
void Draw_Filled_Polygon2D(
    POLYGON2D_PTR poly, // Многоугольник
    UCHAR *vbuffer, // Видеобuffer
    int mempitch); // Шаг памяти
```

// 16-битовая версия

```
void Draw_Filled_Polygon2D16(
    POLYGON2D_PTR poly, // Многоугольник
    UCHAR *vbuffer, // Видеобuffer
    int mempitch); // Шаг памяти
```

#### Назначение

Функции `Draw_Filled_Polygon2D*()` чертят заполненный **n-угольник** в 8- или 16-битовом режиме. Функция получает визуализируемый многоугольник, указатель на видеобuffer и шаг памяти. Обратите внимание, что функция выполняет **визуализацию** относительно точки  $(x_0, y_0)$ , так что убедитесь, что она **инициализирована**. Функция ничего не **возвращает**.

#### Пример использования

```
// Чертим многоугольник в первичном буфере
// в 8-битовом режиме
Draw_Filled_Polygon2D(&poly,
    primary_buffer,
    primary_lpitch);
// Чертим многоугольник в первичном буфере
// в 16-битовом режиме
Draw_Filled_Polygon2D16(&poly,
    primary_buffer,
    primary_lpitch);
```

#### Прототип функции

```
int Translate_Polygon2D(
    POLYGON2D_PTR poly, // Переносимый многоугольник
    int dx, int dy); // Перенос
```

#### Назначение

Функция `Translate_Polygon2D()` переносит начальную точку  $(x_0, y_0)$  многоугольника, не изменяя при этом его вершины. Возвращает в случае успешного **выполнения** **TRUE**.

#### Пример использования

```
// Перенос многоугольника 10,-5
Translate_Polygon2D(&poly, 10, -5);
```

#### Прототип функции

```
int Rotate_Polygon2D(
    POLYGON2D_PTR poly, // Поворачиваемый многоугольник
    int theta); // Угол поворота 0-359
```

#### Назначение

Функция `Rotate_Polygon2D()` поворачивает переданный ей многоугольник против часовой стрелки вокруг начальной точки. Угол должен быть целым числом в диапазоне от 0° до 359°. В случае успешного **выполнения** возвращает **TRUE**.

**Пример использования**  
 // Поворот многоугольника на 10°  
 Rotate\_Polygon2D(&poly, 10);

**Прототип функции**  
 int Scale\_Polygon2D(  
 POLYGON2D\_PTR poly, // Масштабируемый многоугольник  
 float sx, float sy); // Множитель масштабирования

**Назначение**  
 Функция Scale\_Polygon2D() масштабирует переданный многоугольник с коэффициентами масштабирования sx и sy по осям x и y, соответственно. В случае успешного выполнения возвращает TRUE.

**Пример использования**  
 // Увеличение многоугольника в 2 раза  
 Scale\_Polygon2D(&poly, 2, 2);

## Двумерные графические примитивы

**Прототипы функций**  
 int Draw\_Clip\_Line(int x0, int y0, // Начальная точка  
 int x1, int y1, // Конечная точка  
 int color, // Битовый цвет  
 UCHAR \*dest\_buffer, // Видеобуфер  
 int lpitch); // Шаг памяти  
 // 16-битовая версия  
 int Draw\_Clip\_Line16(int x0, int y0, // Начальная точка  
 int x1, int y1, // Конечная точка  
 int color, // 16-битовый цвет  
 UCHAR \*dest\_buffer, // Видеобуфер  
 int lpitch); // Шаг памяти

**Назначение**  
 Функции Draw\_Clip\_Line\*() отсекают прямую при помощи текущего прямоугольника отсечения и чертят ее в буфере. В случае успешного выполнения возвращает TRUE.

**Пример использования**  
 // Черчение прямой во вторичном буфере из точки (10,10) в  
 // точку (100,200) (предполагается 8-битовый режим)  
 Draw\_Clip\_Line(10, 10, 100, 200,  
 5, // Индекс цвета 5  
 back\_buffer,  
 back\_lpitch);  
 // Черчение прямой во вторичном буфере из точки (10,10) в  
 // точку (100,200) (предполагается 16-битовый режим)  
 Draw\_Clip\_Line16(10, 10, 100, 200,  
 RGB16BIT565(0, 31, 0), // Зеленый RGB-цвет  
 back\_buffer,  
 back\_lpitch);

**Прототип функции**  
 int Clip\_Line(int &x1, int &y1, // Начальная точка  
 int &x2, int &y2); // Конечная точка

### Назначение

Функция Clip\_Line() в основном предназначена для внутреннего использования, но вы можете использовать ее для отсечения прямой текущим прямоугольником отсечения. Обратите внимание на то, что функция изменяет переданные ей конечные точки. Функция не выполняет черчение — исключительно математическое вычисление конечных точек. В случае успешного выполнения возвращает TRUE.

### Пример использования

```
// Отсечение прямой, определенной точками x1,y1 и x2,y2
Clip_Line(x1,y1,x2,y2);
```

### Прототипы функций

```
int Draw_Line(int x0, int y0, // Начальная точка
int x1, int y1 // Конечная точка
int color, // 8-битовый цвет
UCHAR *vb_start, // Videобуфер
int lpitch); // Шаг памяти
// 16-битовая версия
int Draw_Line16(int x0, int y0, // Начальная точка
int x1, int y1 // Конечная точка
int color, // 16-битовый цвет
UCHAR *vb_start, // Videобуфер
int lpitch); // Шаг памяти
```

### Назначение

Функции Draw\_Line\*() чертят прямую без отсечения, так что надо убедиться, что конечные точки представляют собой корректные координаты. Эта функция немного быстрее версии с отсечением, поскольку самой операции отсечения здесь нет. В случае успешного выполнения возвращает TRUE.

### Прим. пользования

```
// Черчение прямой во вторичном буфере из точки (10,10) в
// точку (100,200) в 8-битовом режиме
Draw_Line(10,10,100,200,
5, // Цвет 5
back_buffer,
back_lpitch);
// Черчение прямой во вторичном буфере из точки (10,10) в
// точку (100,200) в 16-битовом режиме
Draw_Line16(10,10,100,200,
RGB16BIT(255,255,255), // Белый RGB-цвет
back_buffer,
back_lpitch);
```

### Прототипы функций

```
inline int Draw_Pixel(
int x, int y, // Позиция пикселя
int color, // 8-битовый цвет
UCHAR *video_buffer, // Буфер
int lpitch); // Шаг памяти
// 16-битовая версия
inline int Draw_Pixel16(
int x, int y, // Позиция пикселя
int color, // 16-битовый RGB-цвет
```

```

    UCHAR *video_buffer, // Буфер
    int lpitch); // Шаг памяти

```

#### Назначение

Функции `Draw_Pixel()` чертят одиночный пиксель в данном буфере указанным цветом. В большинстве случаев эта функция не используется, так как это приводит к слишком большим накладным расходам. В случае успешного выполнения возвращает `TRUE`.

#### Пример использования

```

// Выводим пиксель с индексом цвета 100
// в центре экрана 640x480x8
Draw_Pixel(320, 240, 100, back_buffer, back_lpitch);
// Выводим синий пиксель
// в центре экрана 640x480x8
Draw_Pixel16(320, 240, RGB16BIT(0, 0, 31), back_buffer,
    back_lpitch);

```

#### Прототип функции

```

int Draw_Rectangle(
    int x1, int y1, // Верхний левый угол
    int x2, int y2, // Нижний правый угол
    int color, // 8- или 16-битовый индекс цвета
    LPDIRECTDRAW SURFACE7 lpdds);
// Поверхность DirectDraw

```

#### Назначение

Функция `Draw_Rectangle()` чертит прямоугольник на передаваемой в качестве параметра поверхности `DirectDraw`. Учтите, что поверхность должна быть разблокирована. Данная функция использует блиттинг, поэтому она очень быстрая, и работает как в 8-, так и в 16-битовом режиме. В случае успешного выполнения возвращает `TRUE`.

#### Пример использования

```

// Заполнение экрана при помощи блиттинга
Draw_Rectangle(0, 0, 639, 479, lpddsback);

```

#### Прототипы функций

```

void HLine(int x1, int x2, // Начальная и конечная
    // координаты x
    int y, // Координата y прямой
    int color, // 8-битовый цвет
    UCHAR *vbuffer, // Видеобуфер
    int lpitch); // Шаг памяти
void HLine16(int x1, int x2, // Начальная и конечная
    // координаты x
    int y, // Координата y прямой
    int color, // 16-битовый RGB-цвет
    UCHAR *vbuffer, // Видеобуфер
    int lpitch); // Шаг памяти

```

#### Назначение

Функции `HLine()` очень быстро чертят горизонтальную прямую. Функции ничего не возвращают.

#### Пример использования

```

// Чертим горизонтальный отрезок из точки 10,100 в точку
// 100,100 в 8-битовом режиме с индексом цвета 20

```

```
HLine(10, 100, 100, 20, back_buffer, back_lpitch);
// Чертим горизонтальный отрезок из точки 10,100 в точку
// 100,100 в 16-битовом режиме синим цветом
HLine(10, 100, 100, RGB16Bit(0,0,255),
back_buffer, backlpitch);
```

#### Прототипы функций

```
void VLine(int y1, int y2, // Начальная и конечная
// координаты y
int x, // Координата x отрезка
int color, // 8-битовый цвет
UCHAR *vbuffer, // Видеобуфер
int lpitch); // Шаг памяти
void VLine16(int y1, int y2, // Начальная и конечная
// координаты y
int x, // Координата x отрезка
int color, // 16-битовый RGB-цвет
UCHAR *vbuffer, // Видеобуфер
int lpitch); // Шаг памяти
```

#### Назначение

Функции `VLine()` быстро чертят вертикальные отрезки. Эти функции не **настолько** быстры, как функции `HLine()`, но быстрее чем `Draw_Line()`. Функции ничего не возвращают.

#### Пример использования

```
// Чертим отрезок от точки 320,0 в точку 320,479 в
// 8-битовом режиме с индексом цвета 54
VLine(0, 479, 320, 54,
primary_buffer,
primarylpitch);
// Чертим отрезок от точки 320,0 в точку 320,479 в
// 16-битовом режиме зеленым цветом
VLine(0, 479, 320, RGB16Bit(0,255,0),
primary_buffer,
primarylpitch);
```

#### Прототип функции

```
void Screen_Transitions(
int effect // Преобразование экрана
UCHAR *vbuffer, // Видеобуфер
int lpitch); // Шаг памяти
// Команды преобразования экрана
#define SCREEN_DARKNESS 0 // Затухание до черного цвета
#define SCREEN_WHITENESS 1 // Затухание до белого цвета
#define SCREEN_SWIPE_X 2 // Горизонтальное стирание
#define SCREEN_SWIPE_Y 3 // Вертикальное стирание
#define SCREEN_DISOLVE 4 // Растворение пикселей
#define SCREEN_SCRUNCH 5 // Квадратное сжатие
#define SCREEN_BLUENESS 6 // Затухание до синего цвета
#define SCREEN_REDNESS 7 // Затухание до красного цвета
#define SCREEN_GREENNESS 8 // Затухание до зеленого цвета
```

#### Назначение

Функция `Screen_Transition()` выполняет различные преобразования в экранной памяти. Обратите внимание — данные преобразования деструктивны, так что вы должны со-

хранить изображение и/или палитру, если они потребуются вам в дальнейшем. Функция работает только в 8-битовом режиме и ничего не возвращает.

#### Пример использования

```
// Затухание экрана до черного цвета
Screen_Transition(SCREEN_DARKNESS, NULL 0);
```

#### Прототипы функций

```
int Draw_Text_GDI(
    char *text           // Строка с завершающим
                        // нулевым символом
    int x, int y,        // Позиция
    COLORREF color,      // RGB-цвет
    LPDIRECTDRAWSURFACE7 lpdds); // Поверхность

int Draw_Text_GDI(
    char *text           // Строка с завершающим
                        // нулевым символом
    int x, int y,        // Позиция
    int color,           // 8-битовый индекс цвета
    LPDIRECTDRAWSURFACE7 lpdds); // Поверхность
```

#### Назначение

Функции Draw\_Text\_GDI() выводит GDI-текст на переданной поверхности в определенной позиции с указанным цветом в 8- и 16-битовых режимах. Обратите внимание, что поверхность должна быть разблокирована. В случае успешного выполнения возвращает TRUE.

#### Пример использования

```
// Выводим строку цветом RGB(100,100,0);
Draw_Text_GDI("This is a test",100,50,
    RGB16Bit(100,100,0),lpddsprimary);
```

## Математические функции и функции обработки ошибок

Мы вскоре займемся переписыванием математической библиотеки, так что здесь математические функции приведены не более чем для краткого **ознакомления**, чтобы вы не удивлялись, встретив их в некоторых из ранних демонстрационных программ. Здесь же приведены и некоторые функции, которые позволяют выводить сообщения об ошибках или диагностические тексты во время работы игрового процессора.

#### Прототип функции

```
int Fast_Distance_2D(int x, int y);
```

#### Назначение

Функция Fast\_Distance() вычисляет расстояние точки (x,y) от начала координат с использованием аппроксимирующей формулы. Возвращает целое значение, имеющее погрешность до 3.5%. Если вас интересуют технические подробности, то при вычислении используется ряд Тейлора.

#### Пример использования

```
int x1=100,y1=200; // Объект 1
int x2=400,y2=150; // Объект 2
// Расстояние между объектами 1 и 2
int dist = Fast_Distance_2D(x1-x2, y1-y2);
```

#### Прототип функции

```
float Fast_Distance_3D(float x, float y, float z);
```

#### Назначение

Функция `Fast_Distance_3D()` вычисляет расстояние от точки  $(x, y, z)$  до начала координат  $(0, 0, 0)$  с использованием быстрой аппроксимирующей формулы. Погрешность вычисления — до 11%.

#### Пример использования

```
// Вычисляем расстояние от точки (0,0,0) до (100,200,300)
float dist = Fast_Distance_3D(100,200,300);
```

#### Прототип функции

```
int Find_Bounding_Box_Poly2D(
    POLYGON2D_PTR poly, // Многоугольник
    float &min_x, float &max_x, // Ограничивающий
    float &min_y, float &max_y); // прямоугольник
```

#### Назначение

Функция `Find_Bounding_Box_Poly2D()` вычисляет наименьший прямоугольник, содержащий указанный многоугольник. В случае успешного выполнения возвращает `TRUE`. Обратите внимание на передачу параметров по ссылке.

#### Пример использования

```
POLYGON2D poly; // Считаем, что структура инициализирована
int min_x, max_x, // Переменные, в которых хранится
    min_y, max_y; // результат вычислений
// Поиск окружающего прямоугольника
Find_Bounding_Box_Poly2D(&poly, min_x, max_x, min_y, max_y);
```

#### Прототип функции

```
int Open_Error_File(char *filename);
```

#### Назначение

Функция `Open_Error_File()` открывает дисковый файл, в который функцией `Write_Error()` будут записываться сообщения об ошибках. В случае успешного выполнения возвращает `TRUE`.

#### Пример использования

```
// Открываем журнал для записи сообщений об ошибках
Open_Error_File("errors.log");
```

#### Прототип функции

```
int Close_Error_File(void);
```

#### Назначение

Функция `Close_Error_File()` закрывает предварительно открытый файл для сообщений об ошибках. При вызове этой функции в случае, когда файл для сообщений об ошибках не был открыт, никакие действия выполняться не будут. В случае успешного выполнения возвращает `TRUE`.

#### Пример использования

```
// Закрываем файл с сообщениями об ошибках. Обратите
// внимание на отсутствие параметров у данной функции
Close_Error_File();
```

### Прототип функции

```
int Write_Error(char *string, ...); // Форматированная
                                   // строка с сообщением
                                   // об ошибке
```

### Назначение

Функция `Write_Error()` записывает сообщение об ошибке в предварительно открытый при помощи функции `Open_Error_File()` файл. Если файл не открыт, возвращает `FALSE`. Обратите внимание на троеточие в объявлении функции — данную функцию вы можете использовать так же, как и функцию `printf()`. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Записываем различные сообщения
Write_Error("\nSystem Starting...");
Write_Error("\nx-vel = %d", y-vel = %d", xvel, yvel);
```

## Функции для работы с растровыми изображениями

Очередной набор функций предназначен для работы с растровыми изображениями и .BMP-файлами.

### Прототип функции

```
int Load_Bitmap_File(
    BITMAP_FILE_PTR bitmap, // Структура данных для файла
    char *filename);        // Имя загружаемого файла
```

### Назначение

Функция `Load_Bitmap_File()` загружает файл в формате .BMP с диска в структуру `BITMAP_FILE`, переданную в качестве параметра. Функция загружает 8-, 16- и 24-битовые изображения, а также палитру для 8-битового файла. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Загрузка файла "andre.bmp" с диска
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "andre.bmp");
```

### Прототип функции

```
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap);
```

### Назначение

Функция `Unload_Bitmap_File()` освобождает память, выделенную для буфера изображения, загруженного в структуру `BITMAP_FILE`. Эта функция вызывается тогда, когда вы окончили работу с данным изображением. Структура может использоваться повторно, но перед этим необходимо освободить выделенную память. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Закрываем открытый файл изображения
Unload_Bitmap_File(&bitmap_file);
```

### Прототип функции

```
int Create_Bitmap(
    BITMAP_IMAGE_PTR image, // Растровое изображение
    int x, int y,           // Начальная позиция
```

```
int width, int height // Размер изображения
int bpp=8);           // Цветовая глубина
```

### Назначение

Функция `Create_Bitmap()` создает 8- или 16-битовое растровое изображение в системной памяти в указанной позиции с заданным размером. Изначально это пустое изображение. Данное изображение не является поверхностью `DirectDraw`, так что ни ускорение, ни отсечение недоступны. По умолчанию функция создает 8-битовое изображение. В случае успешного выполнения возвращает `TRUE`,

НА ЗАМЕТКУ

`BITMAP_FILE` и `BITMAP_IMAGE` имеют существенное отличие. `BITMAP_FILE` представляет дисковый .BMP-файл, в то время как `BITMAP_IMAGE` — объект в системной памяти.

### Пример использования

```
// Создаем 8-битовое растровое изображение
// размером 64x64 в позиции (0,0)
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 64,64);
// Создаем 16-битовое растровое изображение
// размером 32x32 в позиции (100,100)
BITMAP_IMAGE ship2;
// Обратите внимание на дополнительный параметр 16, который
// указывает глубину цвета вместо используемого по умолчанию
// значения 8
Create_Bitmap(&ship, 0,0, 32,32,16);
```

### Прототип функции

```
int Destroy_Bitmap(BITMAP_IMAGE_PTR image);
```

### Назначение

Функция `Destroy_Bitmap()` используется для освобождения памяти, выделенной при создании объекта `BITMAP_IMAGE`. Вы должны вызвать эту функцию после того, как завершите работу с объектом — как правило, при завершении игры или при уничтожении объекта в игре. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Уничтожение предварительно созданного
// изображения BITMAP_IMAGE
Destroy_Bitmap(&ship);
```

### Прототипы функций

```
int Load_Image_Bitmap(
    BITMAP_IMAGE_PTR image, // Место хранения изображения
    BITMAP_FILE_PTR bitmap, // Загружаемый файл
    int cx, int cy,          // Координаты сканирования
                           // (ячейка или абсолютные
                           // координаты)
    int mode);              // Режим сканирования — ячейки
                           // или абсолютные координаты
                           // 16битовая версия

int Load_Image_Bitmap16(
    BITMAP_IMAGE_PTR image, // Место хранения изображения
    BITMAP_FILE_PTR bitmap, // Загружаемый файл
    int cx, int cy,         // Координаты сканирования
```

```

        // (ячейка или абсолютные
        // координаты)
int mode); // Режим сканирования — ячейки
        // или абсолютные координаты
#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS 1

```

### Назначение

Функции `Load_Image_Bitmap*()` используются для получения изображения из предварительно загруженного файла `BITMAP_FILE` в объект типа `BITMAP_IMAGE`. Файл и изображение должны иметь одну и ту же глубину цвета, и вы должны использовать соответствующую версию функции. Для использования данной функции вы должны загрузить файл `BITMAP_FILE` и создать объект `BITMAP_IMAGE`. После этого вызывается функция, которая получает изображение размера, указанного в `BITMAP_IMAGE`. Имеется два режима работы функции: ячейечный и абсолютный. В первом режиме (`BITMAP_EXTRACT_MODE_CELL`) объект получается исходя из предположения, что все изображения в файле имеют некоторый заданный размер  $m \times n$ , с рамкой толщиной 1 пиксель между ячейками (обычно используются размеры  $8 \times 8$ ,  $16 \times 16$ ,  $32 \times 32$ ,  $64 \times 64$  и т.д.). Посмотрите на содержимое файлов `TEMPLATE*.BMP` на прилагаемом компакт-диске, где содержится ряд таких шаблонов изображений. Ячейки нумеруются слева направо, сверху вниз, начиная от  $(0,0)$ . Второй режим (`BITMAP_EXTRACT_MODE_ABS`) использует абсолютные координаты изображения. Этот метод больше подходит для загрузки из одного файла изображений разных размеров,

### Пример использования

```

// 8-битовый пример

// Предполагается, что исходное изображение имеет размер
// 640x480x8, причем оно представляет собой матрицу ячеек
// размером 8x8, где каждая ячейка имеет размер 32x32. Мы
// хотим загрузить третью ячейку во второй строке
// (ячейка 2,1)

// Загружаем файл в память
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "images.bmp");

// Инициализация изображения
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0, 0, 32, 32);

// Сканирование
Load_Image_Bitmap(&ship, &bitmap_file, 2, 1,
    BITMAP_EXTRACT_MODE_CELL);

// Тот же пример в 16-битовом режиме. Предполагается, что
// исходное изображение имеет размер 640x480x16, причем оно
// представляет собой матрицу ячеек размером 8x8, где каждая
// ячейка имеет размер 32x32, Мы хотим загрузить третью
// ячейку во второй строке (ячейка 2,1)

// Загружаем файл в память
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "images.bmp");

```

```
// Инициализация изображения
BITMAP_IMAGE ship2;
Create_Bitmap(&ship2, 0,0, 32,32,16);
```

```
// Сканирование
Load_Image_Bitmap16(&ship2, &bitmap_file, 2,1,
    BITMAP_EXTRACT_MODE_CELL);
```

Для загрузки того же изображения с использованием абсолютных координат надо не забыть об 1-пиксельной границе с каждой стороны ячейки.

```
// 8-битовый пример
Load_Image_Bitmap(&ship, &bitmap_file,
    2*(32+1)+1,1*(32+1)+1,
    BITMAP_EXTRACT_MODE_ABS);
```

#### Прототипы функций

```
int Draw_Bitmap(
    BITMAP_IMAGE_PTR source_bitmap, // Изображение
    UCHAR *dest_buffer,             // Видеобuffer
    int lpitch,                     // Шаг памяти
    int transparent);               // Прозрачность?
```

// 16-битовая версия

```
int Draw_Bitmap16(
    BITMAP_IMAGE_PTR source_bitmap, // Изображение
    UCHAR *dest_buffer,             // Видеобuffer
    int lpitch,                     // Шаг памяти
    int transparent);               // Прозрачность?
```

#### Назначение

Функции `Draw_Bitmap*()` выводит изображение на переданной поверхности в 8- или 16-битовом режиме с или без прозрачности. Если параметр `transparent` равен 1, включен режим прозрачности, и пиксели с индексом 0 или RGB-цветом (0,0,0) в 16-битовом режиме не копируются. В случае успешного выполнения возвращает TRUE.

#### Пример использования

```
// Выводим изображение корабля во вторичный буфер в
// 8-битовом режиме...
Draw_Bitmap( &ship, back_buffer, back_lpitch, 1);
```

// ...и в 16-битовом режиме

```
Draw_Bitmap16( &ship2, back_buffer, back_lpitch, 1);
```

#### Прототип функции

```
int Flip_Bitmap(
    UCHAR *image, // Биты изображения для
                  // вертикального переворота
    int bytes_per_line, // Байтов на строку
    int height);        // Количество строк
```

#### Назначение

Функция `Flip_Bitmap()` используется внутренне, чтобы перевернуть изображение "вверх ногами" в процессе загрузки изображения. Функция выполняет переворачивание в памяти, строка за строкой. В случае успешного выполнения возвращает TRUE.

### Пример использования

```
// Переворачивание изображения в 8-битовом режиме
Flip_Bitmap(ship->buffer, ship->width, ship->height);

// В 16-битовом режиме нам надо удвоить ширину, поскольку на
// пиксель здесь приходится 2 байта
Flip_Bitmap(ship2->buffer, 2*ship2->width, ship2->height);
```

### Прототип функции

```
int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap,
               int dest_x, int dest_y,
               BITMAP_IMAGE_PTR source_bitmap,
               int source_x, int source_y,
               int width, int height);
```

### Назначение

Функция `Copy_Bitmap()` копирует прямоугольную область из одного изображения в другое. Исходная и целевая области могут быть в одном и том же изображении; однако области не должны перекрываться — в противном случае результат не определен. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Загрузка .BMP-файла в память
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "playfield.bmp");

// Инициализация объекта для хранения игрового поля
BITMAP_IMAGE playfield;
Create_Bitmap(&playfield, 0, 0, 400, 400, 16);

// Загружаем данные
Load_Image_Bitmap16(&playfield, &bitmap_file, 0, 0,
                   BITMAP_EXTRACT_MODE_ABSOLUTE);

// Копируем верхнюю часть изображения в нижнюю
Copy_Bitmap(&playfield, 0, 200,
            &playfield, 0, 0,
            200, 200);
```

### Прототип функции

```
int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);
```

### Назначение

Функция `Scroll_Bitmap()` прокручивает переданное изображение вертикально или горизонтально на величины `dx`, `dy` вдоль осей `x` и `y`, соответственно. Положительные значения означают прокрутку вправо и вниз. В случае успешного выполнения возвращает `TRUE`.

### Пример использования

```
// Скроллинг изображения вправо на 2 пикселя
Scroll_Bitmap(&playfield, 2, 0);
```

## Функции для работы с 8-битовыми палитрами

Данные функции по сути представляют собой интерфейс для работы с 256-цветными палитрами. Они работают только в 8-битовом режиме. Учтите, что в оконном приложении

первые и последние 10 цветов зарезервированы для использования Windows, и вы не имеете права их изменять (в полноэкранном режиме вы имеете право изменять любой цвет).

Фундаментальная структура данных, используемая для хранения цвета, ~ структура `PALETTEENTRY` Win32.

```
typedef struct tagPALETTEENTRY {
    BYTE peRed; // 8 битов красного канала
    BYTE peGreen; // 8 битов зеленого канала
    BYTE peBlue; // 8 битов синего канала
    BYTE peFlags; // Управляющие флаги: PC_EXPLICIT для
                // цветов Windows, PC_NOCOLLAPSE для всех
                // остальных
} PALETTEENTRY;
```

Приступим к рассмотрению конкретных функций.

#### Прототип функции

```
int Set_Palette_Entry(
    int color_index, // Индекс изменяемого цвета
    LPPALETTEENTRY color); // Новый цвет
```

#### Назначение

Функция `Set_Palette_Entry()` используется для изменения одного цвета палитры. Вы просто указываете номер изменяемого цвета в диапазоне 0..255 и указатель на структуру `PALETTEENTRY`, в которой хранится цвет — и это обновление палитры станет видимым в следующем кадре. Кроме того, функция обновляет теневую палитру. Заметим, что это достаточно медленная функция, так что при необходимости обновления всей палитры следует пользоваться функцией `Set_Palette()`. В случае успешного выполнения возвращает `TRUE`,

#### Пример использования

```
// Делаем цвет 0 черным
PALETTEENTRY black = {0,0,0,PC_NOCOLLAPSE};
Set_Palette_Entry(0,&black);
```

#### Прототип функции

```
int Get_Palette_Entry(
    int color_index, // Индекс интересующего цвета
    LPPALETTEENTRY color); // Цвет
```

#### Назначение

Функция `Get_Palette_Entry()` получает информацию о записи текущей палитры. Функция очень быстрая, поскольку получает информацию из теневой палитры в оперативной памяти. Таким образом, данную функцию можно вызывать в любой момент, поскольку она не обращается к аппаратному обеспечению. В случае успешного выполнения возвращает `TRUE`.

#### Пример использования

```
// Получаем цвет с индексом 100
PALETTEENTRY color;
Get_Palette_Entry(100,&color);
```

#### Прототип функции

```
int Save_Palette_To_File(
    char *filename, // Имя файла
    LPPALETTEENTRY palette); // Палитра
```

### Назначение

Функция `Save_Palette_To_File()` сохраняет переданную палитру в ASCII-файле на диске для последующей загрузки или обработки. Функция очень удобна для генерации палитр "на лету" и сохранения их на диске. Предполагается, что указатель указывает на 256-цветную палитру, так что будьте внимательны. В случае успешного выполнения возвращает TRUE.

### Пример использования

```
PALETTEENTRY my_palette[256];    // Считаем, что палитра уже
                                  // создана
```

```
// Сохраняем палитру. Имя может быть любым, но я предпочитаю
// использовать расширение *.pal
Save_Palette_To_File("/palettes/custom1.pal",&my_palette);
```

### Прототип функции

```
int Load_Palette_From_File(
    char *filename,    // Файл
    LPPALETTEENTRY palette); // Палитра
```

### Назначение

Функция `Load_Palette_From_File()` используется для загрузки предварительно сохраненной на диске при помощи функции `Save_Palette_To_File()` 256-цветной палитры. Данная функция не загружает палитру аппаратно — для этого вы должны использовать функцию `Set_Palette()`. В случае успешного выполнения возвращает TRUE.

### Пример использования

```
// Загружаем палитру с диска
Load_Palette_From_File("/palettes/custom1.pal",&my_palette);
```

#### НА ЗАМЕТКУ

При инициализации 256-цветного режима при помощи функции `DDraw_Init()` загружается стандартная палитра с именем `PALDATA2.DAT`.

### Прототип функции

```
int Set_Palette(LPPALETTEENTRY set_palette);
```

### Назначение

Функция `Set_Palette()` загружает переданную палитру в аппаратное обеспечение и обновляет теневую палитру. В случае успешного выполнения возвращает TRUE.

### Пример использования

```
// Загрузка палитры в аппаратное обеспечение
Set_Palette(&my_palette);
```

### Прототип функции

```
int Save_Palette(LPPALETTEENTRY sav_palette);
```

### Назначение

Функция `Save_Palette()` сканирует и сохраняет аппаратную палитру в `sav_palette` для дальнейшего сохранения на диске или обработки. Переменная `sav_palette` должна иметь достаточно места для загрузки 256 цветов.

### Пример использования

```
// Получение текущей палитры DirectDraw
PALETTEENTRY hardware_palette[256];
Save_Palette(&hardware_palette);
```

### Прототип функции

```
int Rotate_Colors(  
    int start_index, // Начальный индекс 0..255  
    int end_index); // Конечный индекс 0..255
```

### Назначение

Функция `Rotate_Colors()` циклически смещает набор цветов, работая непосредственно с аппаратным обеспечением. В случае успешного выполнения возвращает `TRUE`, в противном случае — `FALSE`.

### Пример использования

```
// Смещение всей палитры  
Rotate_Colors(0,255);
```

### Прототип функции

```
int Blink_Colors(  
    int command, // Команда процессора мигания  
    BLINKER_PTR new_light, // Данные  
    int id); // Идентификатор мигания
```

### Назначение

Функция `Blink_Colors()` используется для создания асинхронной анимации палитры. Функция слишком большая, чтобы подробно разбирать ее здесь, так что обратитесь к материалу предыдущей главы.

### Пример использования

Отсутствует

## Вспомогательные функции

### Прототип функции

```
DWORD Get_Clock(void);
```

### Назначение

Функция `Get_Clock()` возвращает текущее показание часов в миллисекундах с момента запуска Windows.

### Пример использования

```
// Получение текущего показания часов  
DWORD start_time = Get_Clock();
```

### Прототип функции

```
DWORD Start_Clock(void);
```

### Назначение

Функция `Start_Clock()` выполняет вызов `Get_Clock()` и сохраняет полученный результат в глобальной переменной для дальнейшего использования функцией `Wait_Clock()`. Возвращает значение показания часов в момент вызова.

### Пример использования

```
// Сохранение показаний часов в глобальной переменной  
Start_Clock();
```

### Прототип функции

```
DWORD Wait_Clock(DWORD count); // Количество миллисекунд  
                                // ожидания
```

### Назначение

Функция `Wait_Clock()` ожидает, когда пройдет переданное ей в качестве параметра количество миллисекунд со времени последнего вызова `Start_Clock()`. Возвращает значение показания часов в момент вызова, но возврат не происходит до тех пор, пока не истечет время ожидания.

### Пример использования

```
// Ждем 30 миллисекунд
Start_Clock();
// Некоторый код...
Wait_Clock(30);
```

НА ЗАМЕТКУ

Позже в книге мы будем использовать высокопроизводительные таймеры Windows с лучшим разрешением.

### Прототип функции

```
int Collision_Test(
    int x1, int y1, // Верхний левый угол объекта 1
    int w1, int h1, // Ширина и высота объекта 1
    int x2, int y2, // Верхний левый угол объекта 2
    int w2, int h2); // Ширина и высота объекта 2
```

### Назначение

Функция `Collision_Test()` проверяет наличие перекрытия переданных ей прямоугольников (прямоугольники могут представлять *все*, что угодно). Возвращает TRUE при перекрытии прямоугольников и FALSE в противном случае.

### Пример использования

```
// Перекрываются ли эти два изображения?
if (Collision_Test(ship1->x, ship1->y,
    ship1->width, ship1->height,
    ship2->x, ship2->y,
    ship2->width, ship2->height))
{ // Да, перекрытие есть
} // if
```

### Прототип функции

```
int Color_Scan(
    int x1, int y1, // Верхний левый угол прямоугольника
    int x2, int y2, // Нижний правый угол прямоугольника
    UCHAR scan_start, // Начальный цвет
    UCHAR scan_end, // Конечный цвет
    UCHAR *scan_buffer, // Сканируемая память
    int scan_pitch); // Шаг памяти
```

### Назначение

Функция `Color_Scan()` представляет другой алгоритм определения столкновений, который сканирует прямоугольник на наличие одного 8-битового значения или последовательности значений в некотором непрерывном диапазоне. Вы можете использовать его для определения того, имеется ли интересующий нас индекс цвета в некоторой области. Естественно, эта функция работает с 8-битовыми изображениями, однако она легко расширяема для 16-битового режима. Возвращает TRUE, если цвет(а) *найден(ы)*.

### Пример использования

```
// Поиск цвета в диапазоне 122-124 включительно
Color_Scan(10,10, 50, 50, 122,124,
    back_buffer, back_pitch);
```

## Процессор для работы с объектами блиттера

Хотя типа `BITMAP_IMAGE` в принципе достаточно практически для всего, что вы только можете придумать, у него есть один серьезный недостаток — он не использует поверхности `DirectDraw`, а следовательно, не использует поддержку аппаратного ускорения. Поэтому я создал тип *BOS (blitter object)*, который очень похож на *спрайт*. Спрайт — это не более чем объект, который вы можете перемещать по экрану (обычно не затрагивая при этом фоновое изображение). В нашем случае это не так, и поэтому я и назвал мой анимационный объект не спрайтом, а объектом блиттера.

Процессор для работы с объектами блиттера в данной книге будет использоваться в очень незначительной степени, но, тем не менее, я бы хотел рассмотреть его, поскольку это хороший пример использования поверхностей `DirectDraw` и полного двумерного ускорения. Начнем со структуры данных для BOB.

```
// Структура объекта блиттера
typedef struct BOB_TYP
{
    int state; // Состояние объекта
    int anim_state; // Переменная состояния анимации
    int attr; // Атрибуты объекта
    float x,y; // Позиция вывода изображения
    float xv,yv; // Скорость объекта
    int width, height; // Ширина и высота
    int width_fill; // Используется внутренне для
    // обеспечения ширины поверхности 8*x
    int bpp; // Битов на пиксель (необходимо для
    // поддержки 8/16-битовых режимов)
    int counter_1; // Обобщенные счетчики
    int counter_2;
    int max_count_1; // Обобщенные пороговые значения
    int max_count_2;
    int varsI[16]; // Стек из 16 целых чисел
    float varsF[16]; // Стек из 16 действительных чисел
    int curr_frame; // Текущий кадр анимации
    int num_frames; // Общее количество кадров анимации
    int curr_animation; // Индекс текущей анимации
    int anim_counter; // Используется при преобразовании
    // времени анимации
    int anim_index; // Индекс элемента анимации
    int anim_count_max; // Количество циклов перед анимацией
    // Последовательность анимации
    int *animations[MAX_BOB_ANIMATIONS];
    // Поверхность DirectDraw
    LPDIRECTDRAW_SURFACE7 images[MAX_BOB_FRAMES];
} BOB, *BOB_PTR;
```

BOB представляет собой графический объект, представленный одной или несколькими поверхностями `DirectDraw` (в соответствии с текущими определениями `tfdefine` — до 64). Вы можете перемещать, выводить, анимировать BOB и настраивать параметры его

движения. BOB выводится с учетом **текущего отсека** DirectDraw, так что отсечение используется вместе с аппаратным ускорением. На рис. 3.10 показан BOB и его **взаимоотношения** с кадрами анимации.

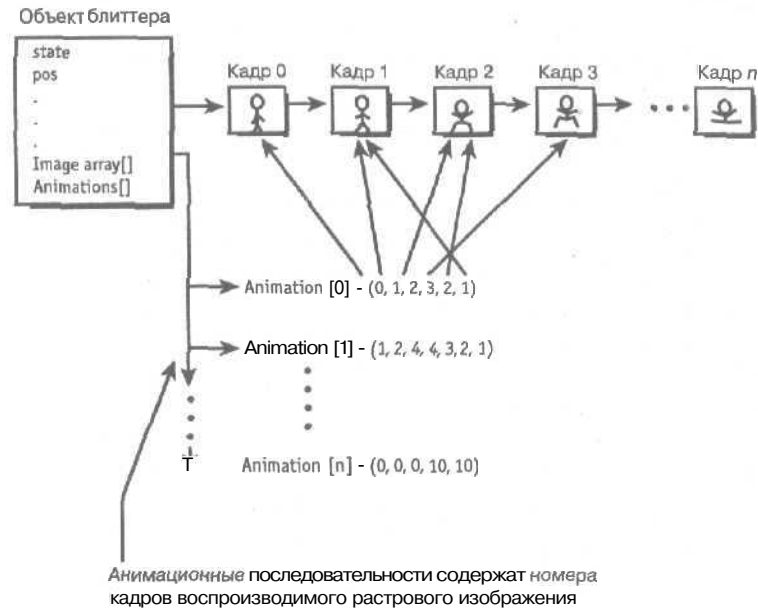


Рис. 3.10. Анимация объекта блиттера

Объекты блиттера поддерживают 8- и 16-битовые изображения и анимационные последовательности, так что вы можете загрузить набор кадров и анимационную последовательность, которая будет воспроизводиться, — что является очень ценным достоинством BOB! Кроме **того**, объекты блиттера сами разбираются, с какой глубиной **цвета** работают, так что единственные функции, которая имеют разные версии для 8- и 16-битового режима — это `Load_Frame_BOB*()` и `Draw_BOB*()`.

Все функции в случае успешного выполнения **возвращают** TRUE, в противном случае - FALSE.

#### Прототип функции

```
int Create_BOB(
    BOB_PTR bob, // Указатель на создаваемый объект
    int x, int y, // Начальная позиция
    int width,
    int height // Размер объекта
    int num_frames, // Общее количество кадров
    int attr, // Атрибуты
    int mem_flags=0, // Флаг памяти поверхности (0 - VRAM)
    USHORT color_key_value=0, // Значение цветового ключа,
    // рассматриваемое либо как 8-битовый
    // индекс, либо как 16-битовое
    // RGB-значение в зависимости от
    // параметра bpp
    int bpp=8); // Количество битов на пиксель
```

### Назначение

Функция **Create\_BOB()** создает и настраивает объект **блиттера**. В дополнение к созданию отдельных поверхностей **DirectDraw** для каждого кадра, она присваивает значения всем внутренним переменным. Большинство параметров функции очевидно, и пояснения требует только параметр **attr.\*** В табл. 3.1 приведены возможные значения атрибутов, которые могут объединяться с использованием побитовой операции **ИЛИ**.

**Таблица 3.1. Атрибуты объекта блиттера**

Значение	Описание
BOB_ATTR_SINGLE_FRAME	Создание BOB с одним кадром
BOB_ATTR_MULTI_FRAME	Создание BOB с несколькими кадрами; при этом анимация BOB представляет собой линейную последовательность кадров 0..n
BOB_ATTR_MULTI_ANIM	Создание BOB с несколькими кадрами и поддержкой последовательностей анимации
BOB_ATTR_ANIM_ONE_SHOT	Если этот флаг установлен, то последовательность анимации будет воспроизведена однократно (определяется внутренней переменной <b>anim_set</b> ). Для повторного воспроизведения требуется сбросить значение данной переменной
BOB_ATTR_BOUNCE	Этот флаг заставляет BOB отражаться при движении от границ экрана. Этот атрибут имеет значение только при вызове функции <b>Move_BOB()</b>
BOB_ATTR_WRAPAROUND	Этот флаг заставляет BOB при движении и пересечении границы экрана появляться с другой стороны. Этот атрибут имеет значение только при вызове функции <b>Move_BOB()</b>

### Пример использования

// Создание 8-битового объекта с одним кадром в позиции  
// (50,100) с размером 96x64:

```
BOB car;           // Объект-автомобиль
                  // Создаем объект
if (!Create_BOB(
    &car, 50,100,    // Объект и его позиция
    96,64           // Размер объекта
    1,             // Количество кадров
    BOB_ATTR_SINGLE_FRAME, // Атрибуты
    0,             // Флаг памяти
    0,             // Цветовой ключ
    8))            // Битов на пиксель
{ /* error */ }
```

// Создание 16-битового объекта с 8 кадрами  
// и размером 32x32:

```
BOB ship;          // Космический корабль
                  // Создаем объект
if (!Create_BOB(
    &ship, 0,0,      // Объект и его позиция
    32,32,          // Размер объекта
```

```

8,          // Количество кадров
BOB_ATTR_MULTI_FRAME, // Атрибуты
0,          // Флаг памяти
0,          // Цветовой ключ
16))       // Битов на пиксель
{ /* error */ }

```

// Создание 8-битового объекта с поддержкой анимационных последовательностей

```

BOBgreeny; // Зеленый человек
// Создание объекта
if (!Create_BOB(&greeny, 0,0,
32,32,32,BOB_ATTR_MULTI_ANIM,0,0,8))
{ /* error */ }

```

#### СОВЕТ

Обратите внимание на наличие у последних трех параметров функции значений по умолчанию. Если значения 0,0,8 вас удовлетворяют, вы можете их не вводить.

### Прототип функции

```

int Destroy_BOB(BOB_PTR bob); // Указатель на уничтожаемый
// объект

```

#### Назначение

Функция `Destroy_BOB()` уничтожает предварительно созданный BOB. Глубина цвета значения не имеет.

#### Пример использования

```

// Уничтожение ранее созданного объекта блиттера
Destroy_BOB(&greeny);

```

### Прототип функции

```

int Draw_BOB(BOB_PTR bob, // Указатель на выводимый объект
LPDIRECTDRAW_SURFACE7 dest); // Поверхность назначения
// 16-битовая версия
int Draw_BOB16(BOB_PTR bob, // Указатель на выводимый объект
LPDIRECTDRAW_SURFACE7 dest); // Поверхность назначения

```

#### Назначение

Функция `Draw_BOB*()` выводит объект блиттера на переданной поверхности `DirectDraw`. BOB выводится в текущей позиции из текущего кадра (определяется параметрами анимации). Необходимо убедиться в том, что используется корректная версия функции, иначе можно получить только половину объекта!

#### ВНИМАНИЕ

Для нормальной работы функции поверхность назначения должна быть НЕ заблокирована.

#### Пример использования

```

// Многокадровый объект блиттера - позиционирование в точке
// (50,50) и вывод первого кадра на вторичную поверхность
BOBship; // Космический корабль
// Создаем 8-битовый объект
if (!Create_BOB(&ship, 0,0,
32,32,8,BOB_ATTR_MULTI_FRAME,0))

```

```

{ /* error */ }

// Загрузка объекта

// Установка позиции и кадра объекта
ship.x = 50;
ship.y = 50;
ship.curr_frame = 0;
// Вывод объекта
Draw_BOB(&ship, lpddsback);

Прототип функции
int Draw_Scaled_BOB(
    BOB_PTR bob,          // Указатель на объект
    int swidth, int sheight, // Новые размеры
    LPDIRECTDRAWSURFACE7 dest); // Поверхность назначения
// 16-битовая версия
int Draw_Scaled_BOB16(
    BOB_PTR bob,          // Указатель на объект
    int swidth, int sheight, // Новые размеры
    LPDIRECTDRAWSURFACE7 dest); // Поверхность назначения

```

#### Назначение

Функция `Draw_Scaled_BOB*()` работает так же, как и `Draw_BOB()`, с тем лишь отличием, что она получает размеры объекта и перед выводом он будет масштабирован. Это очень удобная функция, позволяющая сделать объект выглядящим как трехмерный, что очень важно для трехмерных игр.

#### Пример использования

```

// Пример вывода корабля размером 128x128, несмотря на то,
// что его исходные размеры — 32x32
Draw_Scaled_BOB(&ship, 128, 128, lpddsback);

```

#### Прототип функции

```

int Load_Frame_BOB(
    BOB_PTR bob,          // Указатель на объект
    BITMAP_FILE_PTR bitmap, // Указатель на файл
    int frame,            // Номер кадра
    int cx, int cy,        // Положение ячейки или
                          // абсолютные координаты
    int mode);            // Режим сканирования

// 16-битовая версия
int Load_Frame_BOB16(
    BOB_PTR bob,          // Указатель на объект
    BITMAP_FILE_PTR bitmap, // Указатель на файл
    int frame,            // Номер кадра
    int cx, int cy,        // Положение ячейки или
                          // абсолютные координаты
    int mode);            // Режим сканирования

```

#### Назначение

Функция `Load_Frame_BOB*()` работает идентично функции `Load_Image_Bitmap()`, так что детальное описание данной функции можно посмотреть на стр. 166. Единственное до-

бавление — номер кадра. Например, если вы создали объект, который имеет четыре кадра, вы должны загружать кадры один за другим. Кроме того, вы должны использовать функцию, соответствующему видеорежиму экрана.

#### Пример использования

```
// Загрузка 4 кадров в 16-битовый BOB
//из файла в режиме ячеек

BOB ship; // Объект
// Загружаем кадры 0,1,2,3 из позиций
// (0,0), (1,0), (2,0), (3,0)
for (int index=0; index<4; index++)
    Load_Frame_BOB16(&ship, &bitmap16bit,
        index, index, 0,
        BITMAP_EXTRACT_MODE_CELL);
```

НА ЗАМЕТКУ

Остальные функции работают со структурой данных BOB, так что версии функций для 8- и 16-битовых объектов идентичны.

#### Прототип функции

```
int Load_Animation_BOB(
    BOB_PTR bob, // Объект для загрузки
    int anim_index, // Какая анимация (0..15) загружается
    int num_frames, // Количество кадров анимации
    int *sequence); // Указатель на массив, хранящий
    // последовательность
```

#### Назначение

Функция `Load_Animation_BOB()` требует небольшого пояснения. Функция используется для загрузки одного из 16 внутренних массивов BOB, содержащего последовательность анимации. Каждая последовательность содержит массив индексов или номеров кадров последовательности (см. нижнюю часть рис. 3.10).

#### Пример использования

Допустим, есть объект с восемью кадрами — 0,1,...,7, и у вас есть четыре анимации, определенные следующим образом.

```
int anim_walk[] = {0,1,2,1,0};
int anim_fire[] = {5,6,0};
int anim_die[] = {3,4};
int anim_sleep[] = {0,0,7,0,0};
```

Тогда для загрузки анимации в BOB следует поступить следующим образом.

```
// Создание мультианимационного объекта блиттера
// Создание объекта
if (!Create_BOB(&alien, 0,0, 32,32,8,BOB_ATTR_MULTI_ANIM,0))
    { /* error */ }
```

```
// Загрузка кадров...
```

```
// Загрузка перемещения объекта в анимацию 0
Load_Animation_BOB(&alien, 0,5,anim_walk);
```

```
// Загрузка ведения огня объектом в анимацию 1
```

```
Load_Animation_BOB(&alien, 1, 3, anim_fire);
```

```
// Загрузка гибели объекта в анимацию 2  
Load_Animation_BOB(&alien, 2, 2, anim_die);
```

```
// Загрузка ожидания объекта в анимацию 3  
Load_Animation_BOB(&alien, 3, 5, anim_sleep);
```

После загрузки анимаций можно указать активную анимацию и воспроизвести ее с помощью функции, с которой вы вскоре познакомитесь.

#### Прототип функции

```
int Set_Pos_BOB(BOB_PTR bob, // Указатель на объект  
int x, int y); // Новая позиция
```

#### Назначение

Функция `Set_Pos_BOB()` представляет собой простейший способ изменить позицию объекта блиттера. Она не делает ничего, кроме изменения внутренних координат  $(x, y)$ .

#### Пример использования

```
// Изменение положения объекта  
Set_Pos_BOB(&alien, player_x, player_y);
```

#### Прототип функции

```
int Set_Vel_BOB(BOB_PTR bob, // Указатель на объект  
int xv, int yv); // Новая скорость
```

#### Назначение

Каждый объект блиттера имеет внутренние переменные, определяющие его скорость. Функция `Set_Vel_BOB()` просто присваивает им новые значения. Скорость объекта используется только функцией `Move_BOB()`, однако вы можете использовать значения полей `xv`, `yv` структуры объекта и самостоятельно.

#### Пример использования

```
// Объект движется горизонтально  
Set_Vel_BOB(&alien, 10, 0);
```

#### Прототип функции

```
int Set_Anim_Speed_BOB(BOB_PTR bob, // Указатель на объект  
int speed); // Скорость анимации
```

#### Назначение

Функция `Set_Anim_Speed()` устанавливает новое значение поля `anim_count_max` структуры объекта, определяющего скорость анимации. Чем больше это число, тем медленнее анимация; чем меньше (наименьшее возможное значение — 0), тем анимация быстрее. Это значение влияет только на работу функции `Animate_BOB()`. Конечно же, она имеет смысл только для многокадрового объекта блиттера.

#### Пример использования

```
// Скорость анимации — изменение один раз в 30 кадров  
Set_Anim_Speed_BOB(&alien, 30);
```

#### Прототип функции

```
int Set_Animation_BOB(  
BOB_PTR bob, // Указатель на объект  
int anim_index); // Индекс анимации
```

### Назначение

Функция `Set_Animation_BOB()` устанавливает текущую анимацию, которая будет воспроизводиться.

### Пример использования

```
//Активируем вторую анимационную последовательность
Set_Animation_BOB(&alien, 2);
```

НА ЗАМЕТКУ

Эта функция также сбрасывает анимацию к первому кадру последовательности.

### Прототип функции

```
int Animate_BOB(BOB_PTR bob); // Указатель на объект
```

### Назначение

Функция `Animate_BOB()` анимирует объект блиттера. Обычно данная функция вызывается один раз на кадр для обновления анимации объекта.

### Пример использования

```
// Стираем все изображения...
// Перемещаем все объекты...
// Анимлируем их
Animate_BOB(&alien);
```

### Прототип функции

```
int Move_BOB(BOB_PTR bob); // Указатель на объект
```

### Назначение

Функция `Move_BOB()` перемещает объект в соответствии со значениями скорости объекта `xv`, `yv` (устанавливаемыми функцией `Set_Vel_BOB()`). При этом, в зависимости от атрибутов объекта, происходит отражение от границ экрана, появление при проходе через границы с другой стороны или не выполняется никаких специальных действий. Функция вызывается один раз на кадр, сразу до (или после) функции `Animate_BOB()`.

### Пример использования

```
// Анимлируем объект
Animate_BOB(&alien);
// Перемещаем его
Move_BOB(&alien);
```

### Прототип функции

```
int Hide_BOB(BOB_PTR bob); // Указатель на объект
```

### Назначение

Функция `Hide_BOB()` просто устанавливает равной 0 внутреннюю переменную видимости объекта, после чего функция `Draw_BOB()` ничего не выводит на экран.

### Пример использования

```
// Скрываем объект
Hide_BOB(&alien);
```

### Прототип функции

```
int Show_BOB(BOB_PTR bob); // Указатель на объект
```

### Назначение

Функция `Show_BOB()` устанавливает флаг видимости равным 1 (действие, отменяющее результат выполнения функции `Hide_BOB()`).

#### Пример использования

```
Hide_BOB(&alien);  
// Например, некоторые вызовы GDI  
Show_BOB(&alien);
```

#### Прототип функции

```
int Collision_BOBS(  
    BOB_PTR bob1, // Указатель на первый объект  
    BOB_PTR bob2); // Указатель на второй объект
```

#### Назначение

Функция `Collision_BOBS()` проверяет наличие **перекрытия** прямоугольников, описанных вокруг двух объектов. Может использоваться для определения столкновений в игре, например, для проверки поражения объекта ракетой и в т.п. случаях.

#### Пример использования

```
// Проверка нанесения поражения ракетой:  
if (Collision_BOBS(&missile, &player))  
{ /* Звук взрыва */ }
```

На этом можно считать графический модуль библиотеки **T3DLIB** изученным и перейти к очередному модулю.

## Система ввода DirectX

Написание функций-оболочек вокруг DirectInput — задача если и требующая участия мозга, то в основном спинного. **Все**, что нам надо, ~ это создать API с очень простым интерфейсом и несколькими параметрами. Интерфейс должен поддерживать **следующую** минимальную функциональность:

- инициализация **DirectInput**;
- настройка **захват** клавиатуры, мыши и джойстика;
- чтение данных из любого устройства ввода;
- завершение работы, освобождение устройств и ресурсов.

Я создал соответствующий API, который находится в файлах **T3DLIB2.CPP\H** на прилагаемом компакт-диске. API берет на себя заботу об инициализации **DirectInput** и чтении любого устройства. Перед тем как приступить к его рассмотрению, взгляните на рис. 3.11, где показано взаимодействие устройств и потоков данных.

Глобальные переменные библиотеки

```
LPDIRECTINPUT8 lpd; // Объект DirectInput  
LPDIRECTINPUTDEVICE8 lpdkey; // Клавиатура DirectInput  
LPDIRECTINPUTDEVICE8 lpdmouse; // Мышь DirectInput  
LPDIRECTINPUTDEVICE8 lpdjoy; // Джойстик DirectInput  
GUID joystickGUID; // GUID джойстика  
char joyname[80]; // Имя джойстика  
  
// Весь ввод хранится в этих переменных:  
UCHAR keyboard_state[256]; // Таблица состояния клавиатуры  
DIMOUSESTATE mouse_state; // Состояние мыши  
DIJOYSTATE joy_state; // Состояние джойстика  
int joystick_found; // Наличие джойстика
```

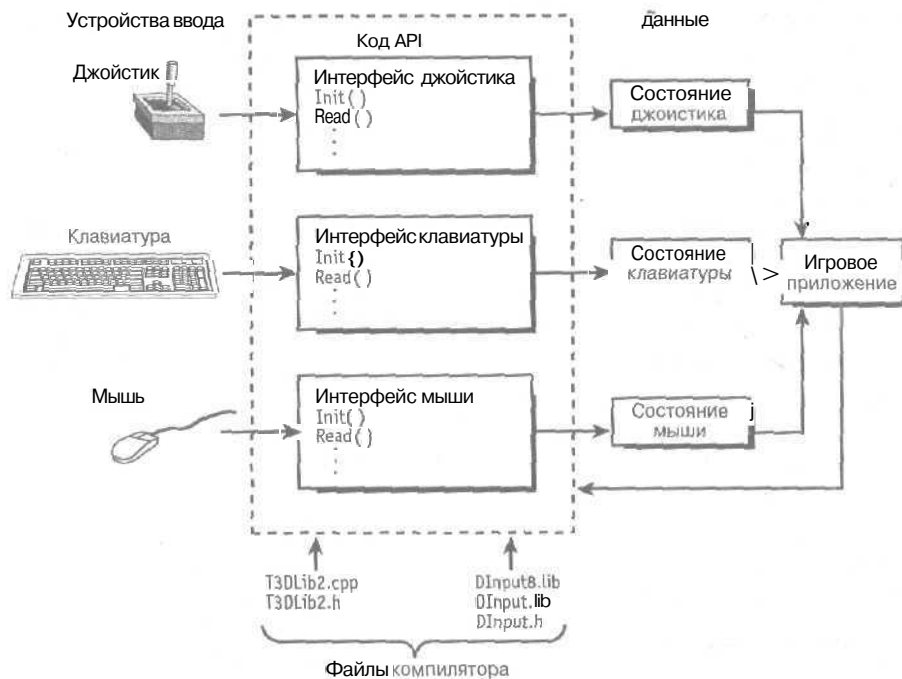


Рис. 3.11. Подсистема *DirectInput*

Весь ввод с клавиатуры помещается в таблицу `keyboard_state[]`, данные, полученные от мыши, хранятся в переменной `mouse_state`, а данные от джойстика размещаются в переменной `joy_state`. Используемые структуры — стандартные структуры устройств Direct Input (за исключением клавиатуры).

```
// Данные мыши
typedef struct DIMOUSESTATE {
    LONG lX;      // Ось x
    LONG lY;      // Ось y
    LONG lZ;      // Ось z
    BYTE rgbButtons[4]; // Состояние кнопок
} DIMOUSESTATE, *LPDIMOUSESTATE;

// Данные джойстика
typedef struct DIJOYSTATE {
    LONG lX;      // Ось x
    LONG lY;      // Ось y
    LONG lZ;      // Ось z
    LONG lRx;     // Поворот вокруг оси x
    LONG lRy;     // Поворот вокруг оси y
    LONG lRz;     // Поворот вокруг оси z
    LONG rgSlider[2]; // Положения ползунка u,v
    DWORD rgdwPOV[4]; // Индикаторы обзора
    BYTE rgbButtons[32]; // Состояние кнопок 0..31
} DIJOYSTATE, *LPDIJOYSTATE;
```

В целом мышь и джойстик грубо эквивалентны — у обоих имеются координаты и состояния кнопок. Переменная `joystick_found` определяет наличие (TRUE) или отсутствие (FALSE) джойстика в системе. А теперь перейдем к рассмотрению функций API.

#### Прототип функции

```
int DInput_Init(void);
```

#### Назначение

Функция `DInput_Init()` инициализирует систему ввода `DirectInput`. Она создает основной COM-объект и возвращает TRUE в случае успешного завершения и FALSE в противном случае и, конечно же, после ее вызова глобальная переменная `lpdi` принимает корректное значение. Функция, однако, не создает никаких устройств.

#### Пример использования

```
if (!DInput_Init())  
{ /* Ошибка инициализации */ }
```

#### Прототип функции

```
void DInput_Shutdown(void);
```

#### Назначение

Функция `DInput_Shutdown()` освобождает все COM-объекты и захваченные во время вызова `DInput_Init()` ресурсы. Обычно данная функция вызывается в самом конце приложения.

#### Пример использования

```
DInput_Shutdown();
```

#### Прототип функции

```
DInput_Init_Keyboard(void);
```

#### Назначение

Функция `DInput_Init_Keyboard()` инициализирует и захватывает клавиатуру. Функция должна всегда корректно работать и возвращать TRUE, кроме ситуации, когда другое приложение `DirectX` некорректно захватило клавиатуру. При успешном завершении глобальная переменная `lpdikey` становится корректным указателем интерфейса.

#### Пример использования

```
if (!DInput_Init_Keyboard()) { /* Ошибка */ }
```

#### Прототип функции

```
int DInput_Init_Mouse(void);
```

#### Назначение

Функция `DInput_Init_Mouse()` инициализирует и захватывает мышь. Функция должна всегда корректно работать и возвращать TRUE, кроме ситуации, когда другое приложение `DirectX` некорректно захватило мышь. При успешном завершении глобальная переменная `lpdimouse` становится корректным указателем интерфейса.

#### Пример использования

```
if (!DInput_Init_Mouse()) { /* Ошибка */ }
```

#### Прототип функции

```
int DInput_Init_Joystick(  
    int min_x=-256, // Минимальное значение x  
    int max_x= 256, // Максимальное значение x  
    int min_y=-256, // Минимальное значение y
```

```
int max_y= 256, // Максимальное значение y
int dead_zone=10); // "Мертвая" зона, проценты
```

#### Назначение

Функция `DInput_Init_Joystick()` инициализирует джойстик. Функция получает пять параметров, которые определяют диапазон значений, передаваемых джойстиком, и "мертвую" зону в процентах. Если вас устраивают значения для диапазонов от -256 до 256, и 10-процентная мертвая зона, вы можете не передавать в функцию никаких параметров. Если функция возвращает TRUE, это значит, что джойстик найден, инициализирован и захвачен. После вызова данной функции указатель на интерфейс `lpdijoy` принимает корректное значение и может использоваться в вашей программе. Кроме того, строка `joynname[]` после вызова функции содержит название используемого джойстика наподобие "Microsoft Side-winder Pro". Ниже приведен пример инициализации джойстика с диапазонами значений от -1024 до 1024 и 5-процентной мертвой зоной.

#### Пример использования

```
if (!DInput_Init_Joystick(-1024, 1024, -1024, 1024, 5))
{ /* Ошибка */ }
```

#### Прототипы функций

```
void DInput_Release_Joystick(void);
void DInput_Release_Mouse(void);
void DInput_Release_Keyboard(void);
```

#### Назначение

Функции `DInput_Release_Joystick()`, `DInput_Release_Mouse()` и `DInput_Release_Keyboard()` освобождают соответствующие устройства ввода по окончании работы с ними. Эти функции могут быть вызваны даже в том случае, если вы не инициализировали соответствующее устройство, так что вы можете безопасно вызывать их в конце вашего приложения. Далее приведен полный пример запуска системы `DirectInput`, инициализации всех устройств, освобождение их и завершение работы.

#### Пример использования

```
// Инициализация DirectInput
DInput_Init();

// Инициализация и захват устройств ввода
DInput_Init_Joystick();
DInput_Init_Mouse();
DInput_Init_Keyboard();

// Цикл ввода .... Вся работа приложения выполняется здесь
// Работа приложения завершена...

// Освобождаем все устройства (порядок не важен)
DInput_Release_Joystick();
DInput_Release_Mouse();
DInput_Release_Keyboard();

// Закрываем DirectInput
DInput_Shutdown();
```

#### Прототип функции

```
int DInput_Read_Keyboard(void);
```

### Назначение

Функция `DInput_Read_Keyboard()` сканирует клавиатуру и помещает данные в массив из 256 элементов типа `BYTE keyboard_state[]`. Это стандартный массив состояния `DirectInput`, так что вы должны использовать константы `DIK_*` при работе с ним. Если клавиша нажата, соответствующее значение массива принимает значение `0x80`. Значения констант `DIK_*` приведены в табл. 3.2.

### Пример использования

```
// Чтение клавиатуры
if (!DInput_Read_Keyboard())
    /* Ошибка */

// Проверка состояния клавиш
if (keyboard_state[DIK_RIGHT])
    /* Перемещаем корабль вправо */
else
    if (keyboard_state[DIK_LEFT])
        /* Перемещаем корабль влево */
```

**Таблица 3.2. Константы состояния клавиатуры DirectInput**

Символ	Описание
<code>DIK_ESCAPE</code>	<Esc>
<code>DIK_0-9</code>	Основная клавиатура: от 0 до 9
<code>DIK_MINUS</code>	-
<code>DIK_EQUALS</code>	=
<code>DIK_BACK</code>	<Backspace>
<code>DIK_TAB</code>	<Tab>
<code>DIK_A-Z</code>	Клавиши от A до Z
<code>DIK_LBRACKET</code>	{
<code>DIK_RBRACKET</code>	}
<code>DIK_RETURN</code>	<Enter> на основной клавиатуре
<code>DIK_LCONTROL</code>	Левая клавиша <Ctrl>
<code>DIK_LSHIFT</code>	Левая клавиша <Shift>
<code>DIK_RSHIFT</code>	Правая клавиша <Shift>
<code>DIK_LMENU</code>	Левая клавиша <Alt>
<code>DIK_SPACE</code>	Клавиша пробела
<code>DIK_F1-15</code>	Функциональные клавиши от 1 до 15
<code>DIK_NUMPAD0-9</code>	Клавиши цифровой клавиатуры
<code>DIK_ADD</code>	Знак + на цифровой клавиатуре
<code>DIK_NUMPADENTER</code>	<Enter> на цифровой клавиатуре
<code>DIK_RCONTROL</code>	Правая клавиша <Ctrl>
<code>DIK_RMENU</code>	Правая клавиша <Alt>

Символ	Описание
<b>DIK_HOME</b>	<Home>
<b>DIK_UP</b>	<t>
<b>DIK_PRIOR</b>	<PgUp>
<b>DIK_LEFT</b>	<←>
<b>DIK_RIGHT</b>	<→>
<b>DIK_END</b>	<End>
<b>DIK_DOWN</b>	<!>
<b>DIK_NEXT</b>	<PgDown>
<b>DIK_INSERT</b>	<Insert>
<b>DIK_DELETE</b>	<Delete>

*Примечание.* Элементы, выделенные жирным шрифтом, означают следование по порядку; например, **DIK\_0-9** означает, что здесь представлены константы **DIK\_0**, **DIK\_1**, **DIK\_2** и т.д.

#### Прототип функции

```
int DInput_Read_Mouse(void);
```

#### Назначение

Функция **DInput\_Read\_Mouse()** считывает состояние мыши и помещает его в переменную **mouse\_state**, представляющую собой структуру **DIMOUSESTATE**. Данные, помещаемые в нее, представлены в относительном дельта-режиме. В большинстве случаев вас будут интересовать только поля **mouse\_state.X**, **mouse\_state.Y** и **rgbButtons[0..2]** (последние представляют собой значения типа **BOOLEAN** для трех кнопок мыши).

#### Пример использования

```
// Чтение мыши
if (!DInput_Read_Mouse())
{ /* Ошибка */ }

// Перемещаем курсор
cx+=mouse_state.X;
cy+=mouse_state.Y;

// Проверяем состояние левой кнопки
if (mouse_state.rgbButtons[0])
    Draw_Pixel(cx,cy,col,buffer,pitch);
```

#### Прототип функции

```
int DInput_Read_Joystick(void);
```

#### Назначение

Функция **DInput\_Read\_Joystick()** опрашивает джойстик и помещает считанные данные в переменную **joy\_state**, которая представляет собой структуру **DIJOYSTATE**. Если джойстика в системе нет, функция **возвращает** значение **FALSE**, а переменная **joy\_state** не получает корректные значения. В случае успешного завершения функции **joy\_state** содержит информацию о состоянии джойстика. Данные координат по осям находятся в заданном ва-

ми ранее диапазоне, а состояния кнопок определяются значениями элементов массива `rgbButtons[]` типа `BOOLEAN`,

#### Пример использования

```
// Чтение данных джойстика
if (!DInput_Read_Joystick())
    { /* Ошибка */ }

// перемещаем корабль
ship_x+=joy_state.X;
ship_y+=joy_state.Y;

// Состояние кнопок
if (joy_state.rgbButtons[0])
    { // Запуск ракет // }
```

Конечно, ваш джойстик может иметь много кнопок и осей. В этом случае вы можете использовать другие поля переменной `joy_state`, определенные в структуре `DJOYSTATE` `DirectInput`.

#### НА ЗАМЕТКУ

Несмотря на то, что мы используем интерфейс `IDIRECTINPUTDEVICE8`, мы не должны использовать структуру `DJOYSTATE2` — она предназначена для устройств с обратной связью.

Теперь мы перейдем к рассмотрению одной из важнейших в современных играх подсистем — звуку и музыке. Используя DirectX и несколько написанных мною функций-оболочек, озвучить игру оказывается очень просто.

## Звуковая и музыкальная библиотека T3DLIB3

Технология работы со звуком и музыкой полностью взята из предыдущей книги. Все, что касается этого вопроса, находится в файлах `T3DLIB3.CPP|H`. Однако при их использовании вы должны не забыть включить в проект библиотеку импорта `DirectSound DSOUND.LIB`. `DirectMusic` представляет собой чистый COM-объект, так что никакой библиотеки импорта для работы с данным компонентом не требуется. Однако вам все же необходимо включить в проект заголовочные файлы `DirectSound` и `DirectMusic`:

- `DSOUND.H` — стандартный заголовочный файл `DirectSound`;
- `DMKSCtrl.H` — этот и остальные файлы нужны для работы с `DirectMusic`;
- `DMUSICI.H`;
- `DMUSICC.H`;
- `DMUSICF.H`.

Теперь можно приступить к рассмотрению основных элементов заголовочного файла `T3DLIB3.H`.

#### НА ЗАМЕТКУ

В DirectX 8.0+ Microsoft объединила `DirectMusic` и `DirectSound` в `DirectAudio`. Я не вижу особого смысла в переходе к работе с `DirectAudio`, так что в данной книге эти компоненты используются раздельно.

## Заголовочный файл

Заголовочный файл T3DLIB3.H содержит типы, макроопределения и объявления глобальных переменных. Вот какие макроопределения `tfdefine` имеются в этом файле.

```
#define DM_NUM_SEGMENTS 64      // Количество midi-сегментов,
                                // которые могут быть кэшированы
                                // в памяти

// Определения состояний midi
#define MIDI_NULL 0            // Объект не загружен
#define MIDI_LOADED 1          // Объект загружен
#define MIDI_PLAYING 2         // Объект загружен и проигрывается
tfdefine MIDI_STOPPED 3        // Объект загружен, но остановлен

tfdefine MAX_SOUNDS 256        // Максимальное количество звуков в
                                // системе

// Состояния объектов цифрового звука
#define SOUND_NULL 0
#define SOUND_LOADED 1
#define SOUND_PLAYING 1
#define SOUND_STOPPED 3
```

Следующие макросы предназначены для того, чтобы помочь преобразовать значения в диапазоне 0..100 в шкалу в децибелах, и для преобразования символов из многобайтовых в юникод.

```
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100-volume)))
```

```
// Преобразование символов в Unicode
#define MULTI_TO_WIDE(x,y) \
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, \
        y, -1, x, MAX_PATH)
```

Теперь пришло время познакомиться с типами, используемыми звуковым процессором.

## Типы

В звуковом процессоре используются только два типа — один для цифрового звука, и второй — для MIDI.

```
// Отдельный образец звука
typedef struct pcm_sound_typ
{
    LPDIRECTSOUNDBUFFER dsbuffer;    // Буфер DirectSound,
                                      // содержащий звук
    int state;                       // Состояние звука
    int rate;                        // Скорость воспроизведения
    int size;                        // Размер
    int id;                          // Идентификатор звука
} pcm_sound, *pcm_sound_ptr;

// MIDI-сегмент DirectMusic
typedef struct DMUSIC_MIDI_TYP
{
    I
```

```

IDirectMusicSegment *dm_segment; // Сегмент
IDirectMusicSegmentState *dm_segstate; // Состояние
int id; // Идентификатор
// сегмента
int state; // Состояние MIDI
} DMUSIC_MIDI, *DMUSIC_MIDI_PTR;

```

Все звуки и MIDI-музыка хранятся в этих двух структурах. А теперь рассмотрим глобальные переменные.

## Глобальные переменные

Библиотека T3DLIB3 содержит ряд глобальных переменных. Следующие глобальные переменные используются DirectSound.

```

LPDIRECTSOUND lpds; // Указатель на интерфейс DirectSound
DSBUFFERDESC dsbd; // Описание DirectSound
DSCAPS dscaps; // Звук DirectSound
HRESULT dsresult; // Результат DirectSound
DSBCAPS dsbcs; // Буфер захвата DirectSound

```

```

pcm_sound sound_fx[MAX_SOUNDS]; // Массив буферов звука
WAVEFORMATEX pcmwf; // Обобщенная структура

```

DirectMusic использует следующие глобальные переменные.

```

// Глобальные переменные DirectMusic
IDirectMusicPerformance *dm_perf; // Диспетчер DirectMusic
IDirectMusicLoader *dm_loader; // Загрузчик DirectMusic

```

```

// Здесь хранятся все MIDI-объекты DirectMusic
DMUSIC_MIDI dm_midi[DM_NUM_SEGMENTS];
int dm_active_id; // Текущий активный
// сегмент MIDI

```

НА ЗАМЕТКУ

Полужирным шрифтом выделены массивы, хранящие звуки и MIDI-сегменты.

Вы не должны непосредственно работать с глобальными переменными — за исключением, возможно, обращения к интерфейсам. Все, что вам надо, за вас **сделает API**.

Библиотека состоит из двух частей — DirectSound и DirectMusic. Начнем с рассмотрения DirectSound.

## API оболочки DirectSound

DirectSound может быть как сложным, так и простым — в зависимости от того, каким образом вы его используете. Если вы хотите разработать API, который способен на все, то вы в конечном счете придете к использованию API DirectSound. Но если вы хотите получить простой API, который позволяет инициализировать DirectSound и загрузить и воспроизвести звук, то можно обойтись всего несколькими функциями. Вот на что способен разработанный мною API для работы со звуком:

- инициализировать DirectSound и завершать его работу;
- загружать WAV-файлы в формате 11 kHz 8 бит моно;
- воспроизводить загруженные звуковые файлы;

- останавливать воспроизведение звуков;
- проверять состояние звука;
- изменять громкость, скорость воспроизведения или стереопанорамирование;
- удалять **звуки** из памяти.

#### НА ЗАМЕТКУ

Если не сказано иное, все функции при успешном завершении возвращают TRUE (1), и FALSE в противном случае.

#### Прототип функции

```
int DSound_Init(void);
```

#### Назначение

Функция `DSound_Init()` используется для инициализации `DirectSound`. Она создает **COM-объект** `DirectSound`, устанавливает уровень приоритета и выполняет другие необходимые действия. Достаточно просто вызвать эту функцию, и ваше приложение готово к воспроизведению звуков.

#### Пример использования

```
if (!DSound_Init(void))
    /* Ошибка */ }
```

#### Прототип функции

```
int DSound_Shutdown(void);
```

#### Назначение

Функция `DSound_Shutdown()` используется для завершения работы `DirectSound` и освобождения всех **COM-интерфейсов**, созданных в процессе работы функции `DSound_Init()`. Однако функция `DSound_Shutdown()` не освобождает память, выделенную для звуков — это вы должны сделать при помощи другой функции.

#### Пример использования

```
if (!DSound_Shutdown())
    /* Ошибка */ }
```

#### Прототипы функций

```
int DSound_Load_WAV(char *filename);
```

#### Назначение

Функция `DSound_Load_WAV()` создает буфер `DirectSound`, загружает звуковой файл в память и подготавливает загруженный звук к воспроизведению. Функция получает полный путь и имя загружаемого файла (включая расширение `.WAV`) и загружает файл с диска. В случае успешного завершения функция **возвращает неотрицательный** идентификатор звука. Вы должны сохранить его, поскольку в дальнейшем **обращаться** к звуку надо будет только с использованием этого идентификатора. Если функция не найдет указанного файла, или в память загружено **слишком** много звуков, она вернет **значение -1**.

#### Пример использования

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
// Проверка на наличие ошибок
if (fire_id == -1)
    /* Ошибка */ }
```

Как именно вы будете хранить полученный идентификатор — полностью **зависит** от вас. Если вас интересуют подробности, то звуковые данные находятся в массиве

`sound_fx[]`, элементы которого имеют тип `pcm_sound`, а в качестве индекса массива используется идентификатор звука.

#### Прототип функции

```
int DSound_Replicate_Sound(int source_id); // Идентификатор
```

#### Назначение

Функция `DSound_Replicate_Sound()` используется для копирования звука без копирования используемой для его хранения памяти. Предположим, например, что у нас есть звук выстрела, и мы хотим выстрелить три раза подряд. Для этого можно загрузить три копии звука выстрела в три разных буфера `DirectSound`, но это приводит к непроизводительному перерасходу памяти. Другое решение — создать дубликат буфера, но без дублирования самих звуковых данных. Вместо копирования мы можем просто использовать указатель на исходные данные.

#### Пример использования

```
int gunshot_ids[8]; // Здесь хранятся все идентификаторы
```

```
// Загрузка основного звука
```

```
gunshot_ids[0] = Load_WAV("GUNSHOT.WAV");
```

```
// Создание копий
```

```
for (int index=1; index<8; index++)
```

```
    gunshot_ids[index] =
```

```
        DSound_Replicate_Sound(gunshot_ids[0]);
```

```
// Теперь вы можете использовать звуки gunshot_ids[0..7]
```

#### Прототип функции

```
int DSound_Play_Sound(
```

```
    int id, // Идентификатор воспроизводимого звука
```

```
    int flags - 0, // 0 или DSBPLAY_LOOPING
```

```
    int volume = 0, // Параметр не используется
```

```
    int rate - 0, // Параметр не используется
```

```
    int pan = 0); // Параметр не используется
```

#### Назначение

Функция `DSound_Play_Sound()` воспроизводит предварительно загруженный звук. Вы просто передаете функции идентификатор звука и флаг, указывающий, следует ли воспроизвести звук один раз (0) или воспроизводить его циклически, непрерывно (`DSBPLAY_LOOPING`). Если звук уже воспроизводится, его воспроизведение будет перезапущено.

#### Пример использования

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
```

```
DSound_Play_Sound(fire_id, 0);
```

#### Прототипы функций

```
int DSound_Stop_Sound(int id);
```

```
int DSound_Stop_All_Sounds(void);
```

#### Назначение

Функция `DSound_Stop_Sound()` используется для остановки воспроизведения единичного звука (если он воспроизводится). Вы просто передаете функции идентификатор звука, и функция прекращает его воспроизведение. Функция же `DSound_Stop_All_Sounds()` останавливает все звуки, воспроизводящиеся в настоящее время.

### Пример использования

```
DSound_Stop_Sound(fire_id);  
// В конце программы имеет смысл на всякий случай остановить  
// воспроизведение всех звуков. Проще всего воспользоваться  
// для этого функцией DSound_Stop_All_Sounds():  
DSound_Stop_All_Sounds();
```

### Прототипы функций

```
int DSound_Delete_Sound(int id); // Идентификатор звука  
int DSound_Delete_All_Sounds(void);
```

### Назначение

Функция `DSound_Delete_Sound()` удаляет звук из памяти и освобождает связанный с ним буфер `DirectSound`. Если данный звук в настоящий момент воспроизводится, то функция начинает с того, что прекращает его проигрывание. Функция `DSound_Delete_All_Sounds()` удаляет все предварительно загруженные звуки.

### Пример использования

```
DSound_Delete_Sound(fire_id);
```

### Прототип функции

```
int DSound_Status_Sound(int id);
```

### Назначение

Функция `DSound_Status_Sound()` проверяет состояние загруженного звука (по его идентификатору). Вы просто передаете идентификатор звука функции, и функция возвращает одно из следующих значений:

- `DSBSTATUS_LOOPING` — звук воспроизводится циклически.
- `DSBSTATUS_PLAYING` — звук воспроизводится однократно.

Если значение, возвращенное функцией `DSound_Status_Sound()`, не является ни одним из перечисленных, звук не воспроизводится. Далее приведен пример ожидания завершения воспроизведения звука с последующим его удалением.

### Пример использования

```
// Инициализация DirectSound  
DSound_DSound_Init();  
// Загрузка звука  
int fire_id = DSound_Load_WAV("FIRE.WAV");  
// Воспроизведение звука  
DSound_Play_Sound(fire_id);  
// Ожидание завершения воспроизведения  
while(DSound_Status_Sound(fire_id) &  
      {DSBSTATUS_LOOPING | DSBSTATUS_PLAYING});  
// Удаление звука  
DSound_Delete_Sound(fire_id);  
// Завершение работы DirectSound  
DSound_DSound_Shutdown();
```

### Прототип функции

```
int DSound_Set_Sound_Volume(int id, // Идентификатор звука  
                             int vol); // Громкость 0-100
```

### Назначение

Функция `DSound_Set_Sound_Volume()` изменяет громкость звука в режиме реального времени. Функции передаются идентификатор звука и значение громкости в диапазоне 0–100.

### Пример использования

```
// Половинная громкость звука
DSound_Set_Sound_Volume(fire_id, 50);
// Вы всегда можете вернуть максимальную громкость:
DSound_Set_Sound_Volume(fire_id, 100);
```

### Прототип функции

```
int DSound_Set_Sound_Freq(
    int id, // Идентификатор звука
    int freq); // Новая частота воспроизведения 0-100000
```

### Назначение

Функция `DSound_Set_Sound_Freq()` изменяет скорость воспроизведения звука. Поскольку все загружаемые звуки должны иметь формат 11 kHz моно, вот как можно удвоить скорость воспроизведения.

### Пример использования

```
DSound_Set_Sound_Freq(fire_id, 22050);
// Скорость воспроизведения можно и уменьшить...
DSound_Set_Sound_Freq(fire_id, 6000);
```

### Прототип функции

```
int DSound_Set_Sound_Pan(
    int id, // Идентификатор звука
    int pan); // Значение баланса от -10000 до 10000
```

### Назначение

Функция `DSound_Set_Sound_Pan()` устанавливает относительную интенсивность звука из правого и левого динамиков. Значение `-10000` приводит к звучанию только левого динамика, а `10000` — правого. Для того чтобы громкость динамиков была одинакова, надо использовать значение 0.

### Пример использования

```
// Звук справа
DSound_Set_Sound_Pan(fire_id, 10000);
```

## API оболочки DirectMusic

API DirectMusic еще проще, чем API DirectSound. Я разработал функции для инициализации DirectMusic, создания всех необходимых COM-объектов и загрузки и воспроизведения MIDI-файлов. Вот список основных функциональных возможностей API:

- инициализация и завершение работы DirectMusic;
- загрузка MIDI-файлов с диска;
- воспроизведение MIDI-файла;
- остановка воспроизведения MIDI-файла;
- проверка текущего состояния MIDI-сегмента;
- автоматическое подключение к DirectSound, если он был инициализирован;
- удаление MIDI-сегмента из памяти.

#### НА ЗАМЕТКУ

Если не сказано иное, все функции при успешном завершении возвращают TRUE (1), и FALSE в противном случае.

### Прототип функции

```
int DMusic_Init(void);
```

### Назначение

Функция `DMusic_Init()` инициализирует `DirectMusic` и создает все необходимые `COM`-объекты. Вы должны вызвать эту функцию до всех прочих вызовов библиотеки `DirectMusic`. Кроме того, если вы хотите использовать в программе `DirectSound`, то вы должны инициализировать `DirectSound` до вызова `DMusic_Init()`.

### Пример использования

```
if (!DMusic_Init())  
    { /* Ошибка */ }
```

### Прототип функции

```
int DMusic_Shutdown(void);
```

### Назначение

Функция `DMusic_Shutdown()` завершает работу подсистемы `DirectMusic`. Она освобождает все `COM`-объекты, и выгружает все ранее загруженные `MIDI`-сегменты. Эта функция должна быть вызвана в конце вашего приложения, но до завершения работы `DirectSound` — если, конечно, вы использовали `DirectSound` в вашей программе.

### Пример использования

```
if (!DMusic_Shutdown())  
    { /* Ошибка */ }  
// Теперь можно завершить работу DirectSound...
```

### Прототип функции

```
int DMusic_Load_MIDI(char *filename);
```

### Назначение

Функция `DMusic_Load_MIDI()` загружает `MIDI`-сегмент в память и выделяет ему память в массиве `midl_ids[]`. Функция возвращает идентификатор загруженного сегмента или `-1` в случае неуспешного завершения. Возвращенный идентификатор используется для обращения к `MIDI`-сегменту в других функциях API.

### Пример использования

```
// Загружаем файлы  
int explode_id = DMusic_Load_MIDI("explosion.mid");  
int weapon_id = DMusic_Load_MIDI("laser.mid");  
// Проверка корректности загрузки  
if (explode_id == -1 || weapon_id == -1)  
    { /* У нас проблемы! */ }
```

### Прототип функции

```
int DMusic_Delete_MIDI(int id);
```

### Назначение

Функция `DMusic_Delete_MIDI()` удаляет предварительно загруженный сегмент из системы (соответствующий переданному идентификатору).

### Пример использования

```
if (!DMusic_Delete_MIDI(explode_id) ||  
    !DMusic_Delete_MIDI(weapon_id))  
    { /* Ошибка */ }
```

#### Прототип функции

```
int DMusic_Delete_All_MIDI(void);
```

#### Назначение

Функция `DMusic_Delete_All_MIDI()` удаляет из системы все MIDI-сегменты одним вызовом.

#### Пример использования

```
// Удаляем все загруженные сегменты
if (!DMusic_Delete_All_MIDI())
{ /* Ошибка */ }
```

#### Прототип функции

```
int DMusic_Play(int id);
```

#### Назначение

Функция `DMusic_Play()` воспроизводит MIDI-сегмент, идентификатор которого передается функции в качестве параметра.

#### Пример использования

```
// Загружаем файл
int explode_id = DMusic_Load_MIDI("explosion.mid");
// Воспроизводим его
if (!DMusic_Play(explode_id))
{ /* Ошибка */ }
```

#### Прототип функции

```
int DMusic_Stop(int id);
```

#### Назначение

Функция `DMusic_Stop()` останавливает воспроизведение сегмента, идентификатор которого передан функции в качестве параметра. Если сегмент уже остановлен и не воспроизводится, функция не выполняет никаких действий.

#### Пример использования

```
// Останавливаем воспроизведение
if (!DMusic_Stop(weapon_id))
{ /* Ошибка */ }
```

#### Прототип функции

```
int DMusic_Status_MIDI(int id);
```

#### Назначение

Функция `DMusic_Status()` проверяет состояние MIDI-сегмента, идентификатор которого передается функции в качестве параметра. Имеются следующие коды состояния.

```
#define MIDI_NULL 0 // Объект не загружен
#define MIDI_LOADED 1 // Объект загружен
#define MIDI_PLAYING 2 // Объект загружен и воспроизводится
#define MIDI_STOPPED 3 // Объект загружен, но остановлен
```

#### Пример использования

```
// Основной цикл игры
while(1)
{
    if (DMusic_Status(explode_id) == MIDI_STOPPED)
        game_state = GAME_MUSIC_OVER;
} // while
```

Вот и все, что касается API DirectSound и DirectMusic. Как я говорил, позже вы увидите примеры использования этих API. В настоящий момент я хочу представить вашему вниманию окончательную версию консоли игры T3D.

В настоящий момент у нас имеется три основных CPP|H модуля, составляющих библиотеку T3D:

- T3DLIB1.CPP|H — DirectDraw и графические алгоритмы;
- T3DLIB2.CPP|H — DirectInput;
- T3DLIB3.CPP|H — DirectSound и DirectMusic.

Для компиляции программ с использованием указанных библиотек необходимо включить в их исходные тексты указанные заголовочные файлы, а в проект — файлы с исходными текстами функций.

## Окончательная версия консоли игры

В настоящий момент у нас более чем достаточно материала для реализации нашего виртуального графического интерфейса, включая поддержку звука и устройств ввода. Наша цель — реализация графической системы с двойной буферизацией, поддерживающей 8- и 16-битовую оконную и полноэкрannую графику.

Следующий шаг состоит в создании шаблона, основанного на рассмотренной ранее альфа-версии консоли игры. Однако, перед тем как приступить к этому шагу, рассмотрим функционирование виртуального графического интерфейса, и его отображение на реальный графический интерфейс в T3DLIB1.CPP. Звук и музыка существенно проще, поэтому мы не будем рассматривать их отдельно. Вы сможете ознакомиться с функционированием звука, музыки, устройств ввода в демонстрационных программах.

## Графика — реальная и виртуальная

Ранее в этой главе мы уже рассматривали вопросы минимального обобщенного графического интерфейса, необходимого для программной реализации трехмерной графики на произвольном компьютере. Вернемся к этому же вопросу, но уже со всем необходимым инструментарием в руках.

### Запуск системы

Мы предполагаем наличие в графическом интерфейсе виртуального компьютера функции Create\_Window() со следующим прототипом.

```
Create_Window(int width, int height int bit_depth);
```

Эта функция берет на себя все заботы о настройке графической системы, включая открытие окна необходимого размера с указанной глубиной цвета. В нашей реальной реализации эта функциональность разделена на два разных вызова функции, что связано с нашим стремлением отделить функциональность Windows и DirectX. Первая функция представляет собой вызов стандартной функции Windows для создания обычного окна.

А теперь внимание; если вы создаете полноэкрannое приложение, то должны использовать флаг WM\_POPUP, но если приложение будет оконным, то вам потребуется другой флаг, наподобие WM\_OVERLAPPEDWINDOW. Следовательно, настройка графической системы представляет собой двухступенчатый процесс.

1. Выполняем вызов функции Win32 API `Create_Window()` с параметрами, соответствующими полноэкранному или оконному приложению. В полноэкранном приложении *обычно* не используются никакие *управляющие* элементы.
2. Вызов функции-оболочки DirectX, которая получает дескриптор окна (хранящийся в глобальной *переменной*) и все необходимые параметры для завершения работы по настройке графической системы.

Основная работа выполняется на втором шаге. Именно здесь *инициализируется* `DirectDraw`, создаются буферы кадров, генерируется палитра для 8-битового режима, *присоединяются* отсекатели...

Поскольку мы намерены использовать шаблон консоли игры, наш план состоит в том, чтобы в функции `WinMain()` создавалось окно, а затем производился вызов функции `Game_Init()`, которая, в свою очередь, вызывает `DDraw_Init()`, выполняющую всю черновую работу. Напомним прототип этой функции.

```
int DDraw_Init(int width, int height, int bpp,
               int windowed = 0);
```

Как видите, ничего сложного. В виртуальном компьютере все это достигается при помощи одного вызова `Create_Window()`; в реальности эта функциональность реализуется при помощи двух вызовов — один из них создает окно `Windows`, а второй — *инициализирует* `DirectDraw` и присоединяет его к окну.

## Глобальные отображения

Первое, чем мы занимаемся в нашем виртуальном графическом интерфейсе, — это два буфера кадров: видимый и внеэкранный. Мы называем их первичным и вторичным видеобуферами (рис. 3.12).

Эти буферы являются линейно адресуемыми при любом разрешении и глубине цвета (при этом шаг памяти для перехода от одной строки к другой может не совпадать с количеством пикселей в строке, так что нам нужна отдельная переменная, отслеживающая данную величину). Вот какие имена переменных *используются* для виртуальных буферов и их шагов памяти.

```
UCHAR *primary_buffer; // Первичный буфер
int primary_pitch;     // Шаг памяти в байтах
```

```
UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch;     // Шаг памяти в байтах
```

Наша реальная библиотека `T3DLIB1.CPP` по сути идентична сказанному. В ней для указанных целей используются следующие переменные.

```
LPDIRECTDRAWSURFACE7 lpddsprimary; // Первичная поверхность
LPDIRECTDRAWSURFACE7 lpddsback;    // Вторичная поверхность
```

```
UCHAR *primary_buffer; // Первичный видеобуфер
UCHAR *back_buffer;    // Вторичный видеобуфер
int primary_pitch;     // Шаг памяти
int back_pitch;        // Шаг памяти
```

Обратите внимание на две дополнительные переменные, связанные с использованием `DirectX`. Это указатели на поверхности `DirectDraw` для первичной и вторичной поверхностей, использующиеся в ряде вызовов функций, так что без них вам не обойтись.

Кроме того, ширина и высота первичного и вторичного буферов всегда совпадают. Хотя клиентская область первичного буфера может быть окном, а не всем рабочим сто-

лом, вторичный буфер *всегда* имеет тот же размер, что и клиентская область. Вывод — видимый первичный буфер и невидимый вторичный всегда имеют одинаковый размер и глубину цвета.

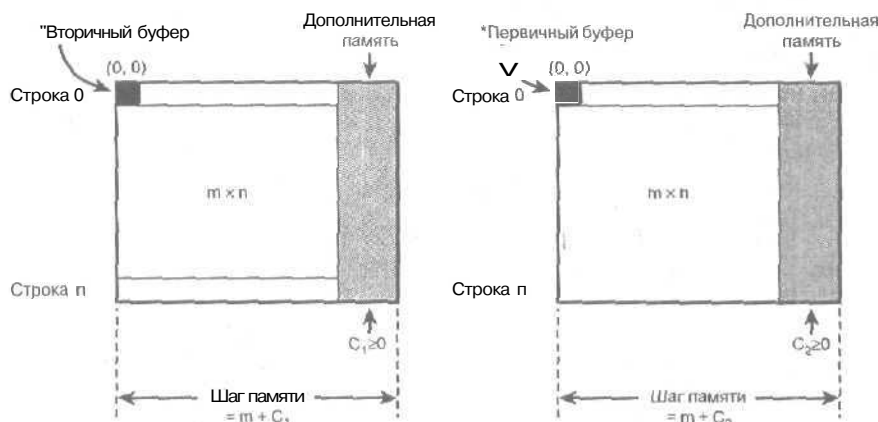


Рис. 3. }2. Буферы кадров

## 256-цветный режим

Хотя вы уже познакомились с функциями для работы с 256-цветными палитрами из библиотеки **T3DLIB1**, я хочу еще раз вернуться к этому вопросу. Итак, вот какие функции предназначены для работы с палитрой.

```
int Set_Palette_Entry(int color_index,
    LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index,
    LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename,
    LPPALETTEENTRY palette);

int Save_Palette_To_File(char *filename,
    LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
```

Здесь выделены функции, представляющие наибольший интерес. Они используются для изменения либо одной записи палитры, либо всей палитры как единого целого. В большинстве случаев изменять по одной записи палитры весьма неэффективно, так что лучше сделать все необходимые изменения в памяти и изменить всю палитру целиком. Функциям передается указатель либо на одну запись **PALETTEENTRY**, либо на массив таких записей. Чтобы освежить вашу память, я приведу определение этой структуры еще раз.

```
typedef struct tagPALETTEENTRY{
    BYTE peRed; // 8 битов красного канала
    BYTE peGreen; // 8 битов зеленого канала
    BYTE peBlue; // 8 битов синего канала
    BYTE peFlags; // Управляющие флаги: PC_EXPLICIT для
                // цветов Windows, PC_NOCOLLAPSE для всех
                // остальных
} PALETTEENTRY;
```

И последнее — если вы выбираете 256-цветный режим, будет загружена палитра `PALDATA2.PAL`. Я предпочитаю использовать эту палитру как хорошо покрывающую все цветовое пространство. Конечно, вы всегда можете загрузить палитру с диска, или палитру, связанную с растровым изображением.

## Функции блокировки

Взглянем на четыре функции, которые необходимы нам для блокирования и разблокирования первичной и вторичной поверхностей. В интерфейсе виртуального компьютера они выглядят следующим образом.

```
Lock_Primary(UCHAR **primary_buffer, int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);
```

```
Lock_Secondary(UCHAR **secondary_buffer,
               int *secondary_pitch);
Unlock_Secondary(UCHAR *secondary_buffer);
```

В реальной графической библиотеке необходимая функциональность реализуется следующими функциями.

```
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);
```

```
UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

Единственное отличие реальных функций от виртуальных в том, что они работают с глобальными переменными

```
UCHAR *primary_buffer; // Первичный буфер
int primary_pitch; // Шаг памяти буфера
```

```
UCHAR *secondary_buffer; // Вторичный буфер
int secondary_pitch; // Шаг памяти буфера
```

а потому не требуют передачи им каких-либо параметров.

## Функции анимации

Последняя функция, о которой я хочу упомянуть — это функция, переключающая первичный и вторичный буферы (или, в некоторых случаях, копирующая содержимое вторичного буфера в первичный). Если вы помните, в интерфейсе виртуального компьютера она имеет имя `Flip_Display()`. В нашей реальной графической библиотеке этим занимается функция `int DDraw_Flip(void)`;

Единственное отличие — в имени функции. Выполняемые ею действия в точности те же, что и в виртуальной модели. Данная функция эффективно справляется со своими обязанностями как в полноэкранном, так и в оконном режимах.

Теперь, когда мы разобрались с тем, какие реальные функции соответствуют функциям виртуальной модели, можно приступить к главной задаче — созданию окончательного варианта консоли игры. Я не поклонник больших кусков кода в книге, но код консоли игры достаточно важен, чтобы привести его полностью.

## Консоль игры — окончательный вариант

Теперь вы достаточно подготовлены к тому, чтобы не только увидеть окончательный вариант консоли игры, но и разобраться в нем. Именно этот шаблон используется в дальней-

шем в качестве основы всех демонстрационных программ книги. Конечно, это не мертвая схема, и вы можете изменять ее, добавляя собственные возможности, но в принципе консоль содержит всю необходимую для создания полноценной игры функциональность.

**НА ЗАМЕТКУ**

Вы можете спросить— что случилось с T3DCONSOLE.CPP|EXE? В предыдущей книге мною была создана консоль игры с таким именем, но, поскольку данную книгу можно рассматривать как продолжение книги Программирование игр для Windows. *Советы профессионала*, я предпочел использовать последовательную нумерацию и добавить 2 в имя файла новой консоли игры — T3DCONSOLE2.CPP|EXE.

```
// T3DCONSOLE2.CPP - шаблон консоли игры. Этот шаблон можно
// использовать для любого приложения— достаточно только
// изменить в нем некоторые параметры, наподобие разрешения
// экрана, указания оконности или полноэкранное™
// приложения, выбора устройств ввода и т.п. В настоящее
// время приложение представляет собой оконное приложение с
// экраном 640x480x16. Таким образом, для работы данного
// приложения вы должны находиться в 16-цветном режиме. Если
// вы хотите получить полноэкранное приложение, вам надо
// только изменить значение WINDOWED_APP на FALSE (0). Для
// использования другой глубины цвета вам надо изменить
// параметры вызова функции DDraw_Init() в функции
// Game_Init()

// ВАЖНО!
// При компиляции убедитесь, что в проект включены
// библиотеки DDRAW.LIB, DSOUND.LIB, DINPUT.LIB и WINMM.LIB,
// а также модули T3DLIB1.CPP, T3DLIB2.CPP и T3DLIB3.CPP, а
// заголовочные файлы T3DLIB1.H, T3DLIB2.H и T3DLIB3.H
// находятся в рабочем каталоге компилятора

// Заголовочные файлы //////////////////////////////////////

#define NITGUID // Делает доступными все COM-интерфейсы.
// Вместо этого в проект можно включить
// библиотеку DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h> // Поддержка функциональности Windows
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // Поддержка функциональности C/C++
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
```

```

#include <ddraw.h> // DirectX
#include <dsound.h>
#include <dmsctrl.h>
#include <dmusicl.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h" // Библиотека игры
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// Макроопределения //////////////////////////////////////

// Интерфейс Windows
#define WINDOW_CLASS_NAME "WIN3DCLASS" // Имя класса
#define WINDOW_TITLE "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH 640 // Размер окна
#define WINDOW_HEIGHT 480

#define WINDOW_BPP 16 // Глубина цвета (8, 16 и т.д.)
// Примечание: в случае оконного приложения глубина цвета
// должна соответствовать системной глубине цвета

#define WINDOWED_APP 1 // 0 - полноэкранное
// приложение, 1 - оконное

// Прототипы //////////////////////////////////////

// Консоль игры
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// Глобальные переменные //////////////////////////////////////

HWND main_window_handle = NULL; // Дескриптор окна
HINSTANCE main_instance = NULL; // Экземпляр
char buffer[256]; // Для вывода текста

// Функции //////////////////////////////////////

LRESULTCALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    // Главный обработчик сообщений
    PAINTSTRUCT ps; // Используется WM_PAINT
    HDC hdc; // Дескриптор контекста устройства

    // Тип сообщения
    switch(msg)
    {
        case WM_CREATE:

```

```

        |
        // Инициализация
        return(0);
    } break;

case WM_PAINT:
    {
        // Начало рисования
        hdc = BeginPaint(hwnd,&ps);

        // Конец рисования
        EndPaint(hwnd,&ps);
        return(0);
    } break;

case WM_DESTROY:
    {
        // Завершение приложения
        PostQuitMessage(0);
        return (0);
    } break;

default: break;

} // switch

// Обработка остальных сообщений
return (DefWindowProc(hwnd, msg, wParam, lParam));

} // WinProc

// WinMain ////////////////////////////////////////

int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX wclass; // Класс окна
    HWND hwnd; // Дескриптор окна
    MSG msg; // Сообщение
    HDC hdc; // Контекст графического устройства

    // Заполнение структуры класса
    wclass.cbSize = sizeof(WNDCLASSEX);
    wclass.style = CS_DBLCLKS | CS_OWNDC |
                  CS_HREDRAW | CS_VREDRAW;
    wclass.lpfnWndProc = WindowProc;
    wclass.cbClsExtra = 0;
    wclass.cbWndExtra = 0;
    wclass.hInstance = hinstance;
    wclass.hIcon = LoadIcon(NULL,
                            IDI_APPLICATION);
    wclass.hCursor = LoadCursor(NULL, IDC_ARROW);

```

```

winclass.hbrBackground - (HBRUSH)
    GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = WINDOW_CLASS_NAME;
winclass.hIconSm - LoadIcon(NULL,
    IDI_APPLICATION);

// Регистрация класса окна
if (!RegisterClassEx(&winclass))
    return(0);

// Создание окна
if (!(hwnd = CreateWindowEx(
    NULL, // Расширенный стиль
    WINDOW_CLASS_NAME, // Класс
    WINDOW_TITLE, // Заголовок
    (WINDOWED_APP ? (WS_OVERLAPPED |
        WS_SYSMENU | WS_CAPTION) :
        (WS_POPUP | WS_VISIBLE)),
    0,0, // Начальные координаты x,y
    WINDOW_WIDTH, WINDOW_HEIGHT, // Ширина, высота
    NULL, // Дескриптор родителя
    NULL, // Дескриптор меню
    hinstance, // Экземпляр приложения
    NULL))) // Дополнительные параметры
    return (0);

// Сохранение в глобальных переменных
main_window_handle = hwnd;
main_instance = hinstance;

// Изменение размеров окна
if (WINDOWED_APP)
{
    // Изменяем размер окна таким образом, чтобы
    // клиентская область имела запрошенный размер
    // (учет границ окна и управляющих элементов)
    RECT window_rect = {0,0,WINDOW_WIDTH-1,
        WINDOW_HEIGHT-1};

    // Вызов для изменения window_rect
    AdjustWindowRectEx(&window_rect,
        GetWindowStyle(main_window_handle),
        GetMenu(main_window_handle) != NULL,
        GetWindowExStyle(main_window_handle));

    // Сохранение глобальных переменных, необходимых для
    // работы DDRAW_Flip()
    window_client_x0 = -window_rect.left;
    window_client_y0 = -window_rect.top;

    // Изменение размеров окна
    MoveWindow(main_window_handle,
        0, // Координатах

```

```

    0, // Координата y
    window_rect.right - window_rect.left, // Ширина
    window_rect.bottom - window_rect.top, // Высота
    FALSE);

    // Вывод окна
    ShowWindow(main_window_handle, SW_SHOW);
} // if windowed

// Инициализация консоли игры
Game_Init();

// Запрет реакции на CTRL-ALT-DEL, ALT-TAB.
// Закомментируйте эту строку, если она вызывает крах
// системы
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);

// Вход в главный цикл событий
while(1)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // Проверка на запрос выхода
        if (msg.message == WM_QUIT)
            break;

        // Преобразование клавиш
        TranslateMessage(&msg);

        // Передача сообщения обработчику
        DispatchMessage(&msg);
    } // if

    // Главная функция игры
    Game_Main();
} // while

// Завершение игры и освобождение ресурсов
Game_Shutdown();

// Разрешение реакции на CTRL-ALT-DEL, ALT-TAB.
// Закомментируйте эту строку, если она вызывает крах
// системы
SystemParametersInfo(SPI_SCREENSAVERRUNNING, FALSE, NULL, 0);

// Выход в Windows
return(msg.wParam);
} // WinMain

// Функции консоли игры T3D II //////////////////////////////////////
int Game_Init(void *parms)
{
    // В этой функции выполняется вся инициализация игры

```

```

// Настройка DirectDraw (измените параметры на
// необходимые вам)
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT,
           WINDOW_BPP, WINDOWED_APP);

// Инициализация DirectInput
DInput_Init();

// Захват клавиатуры
DInput_Init_Keyboard();

// Здесь могут быть вызовы для захвата других устройств
// ввода...

// Инициализация DirectSound и DirectMusic
DSound_Init();
DMusic_Init();

// Скрытие мыши
if (!WINDOWED_APP)
    ShowCursor(FALSE);

// Инициализация генератора случайных чисел
srand(Start_Clock());

// Другой код инициализации...
\

// Успешное завершение
return(1);
} // Game_Init

////////////////////////////////////

int Game_Shutdown(void *parms)
{
    // Эта функция завершает работу игры и освобождает все
    // захваченные ресурсы

    // Освобождение всех захваченных вами ресурсов....

    // Завершение работы DirectMusic
    DMusic_Delete_All_MIDI();
    DMusic_Shutdown();

    // Завершение работы DirectSound
    DSound_Stop_All_Sounds();
    DSound_Delete_All_Sounds();
    DSound_Shutdown();

    // Освобождение захваченных устройств ввода
    DInput_Release_Keyboard();

    // Завершение работы DirectInput
    DInput_Shutdown();

```

```

// Завершение работы DirectDraw
DDraw_Shutdown();

// Успешное завершение
return(1);
} // Game_Shutdown

////////////////////////////////////

int Game_Main(void *parms)
{
    // Эта функция постоянно вызывается в реальном времени и
    // в ней располагается вся необходимая функциональность
    // игры.

    int index; // Переменная цикла

    // Запуск таймера
    Start_Clock();

    // Очистка поверхности вывода
    DDraw_Fill_Surface(lpddsback, 0);

    // Считывание клавиатуры и других устройств ввода
    DInput_Read_Keyboard();

    // Логика игры...

    // Переключение поверхностей
    DDraw_FLip();

    // Синхронизация для достижения скорости 30 fps
    Wait_Clock(30);

    // Проверка на запрос окончания игры
    if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
    {
        PostMessage(main_window_handle, WM_DESTROY, 0, 0);
    } // if

    // Успешное завершение
    return(1);
} // Game_Main

////////////////////////////////////

```

Как видите, в листинге выделены некоторые особо важные для понимания фрагменты. Сейчас мы рассмотрим их более детально.

### Открытие окна консоли игры

Первые выделенные строки кода относятся к разделу макроопределений.

```

#define WINDOW_WIDTH    640    // Размер окна
#define WINDOW_HEIGHT   480

```

```
#define WINDOW_BPP    16 // Глубина цвета (8,16 и т.д.)
//Примечание: в случае оконного приложения глубина цвета
// должна соответствовать системной глубине цвета

#define WINDOWED_APP  1 //0 - полноэкранное
                        //приложение, 1 - оконное
```

Это весьма важная часть кода, которая управляет размером окна (или размером экрана), глубиной цвета и видом приложения (оконное или полноэкранное). В настоящий момент это — оконное приложение размером 640×480 и глубиной цвета 16 битов на пиксель. Указанные параметры используются в ряде мест в коде приложения. Самые главные применения указанных параметров — при создании окна в функции WinMain() и в вызовах DDraw\_Init() и GameMain(). Начнем с рассмотрения WinMain().

Если вы взглянете на вызов CreateWindow() в функции WinMain(), то увидите, что в ней используется тернарный оператор ?:, проверяющий, является ли данное приложение оконным.

```
// Создание окна
if (!hwnd = CreateWindowEx(
    NULL,           // Расширенный стиль
    WINDOW_CLASS_NAME, // КЛАСС
    WINDOW_TITLE,   // Заголовок
    (WINDOWED_APP ? (WS_OVERLAPPED |
                     WS_SYSMENU | WS_CAPTION) :
     (WS_POPUP | WS_VISIBLE)),
    0,0,           // Начальные координаты x,y
    WINDOW_WIDTH, WINDOW_HEIGHT, // Ширина, высота
    NULL,         // Дескриптор родителя
    NULL,         // Дескриптор меню
    hinstance,    // Экземпляр приложения
    NULL))) // Дополнительные параметры
    return(0);
```

Таким образом, функция создает окно с соответствующими режиму приложения флагами Windows. Если это оконное приложение, следовательно, у окна должны быть рамка и ряд управляющих элементов, поэтому для оконного приложения используются флаги WS\_OVERLAPPED | WS\_SYSMENU | WS\_CAPTION.

С другой стороны, при работе в полноэкранном режиме размер окна равен размеру поверхности DirectDraw, но без управляющих элементов. Следовательно, должен быть использован стиль окна WM\_POPUP.

Код после создания окна достаточно интересен. Он изменяет размер окна таким образом, чтобы клиентская область имела запрашиваемый размер (а не размер, меньший на величину размера рамок и управляющих элементов).

Вспомните программирование в Windows — когда вы создаете окно с размером WINDOW\_WIDTH×WINDOW\_HEIGHT, это не означает, что клиентская область также имеет размер WINDOW\_WIDTH×WINDOW\_HEIGHT. Это — размер всей области окна. Таким образом, если у окна нет рамок и управляющих элементов, размер клиентской области совпадает с запрошенным размером; если же они есть — размер оказывается несколько меньшим (рис. 3.13).

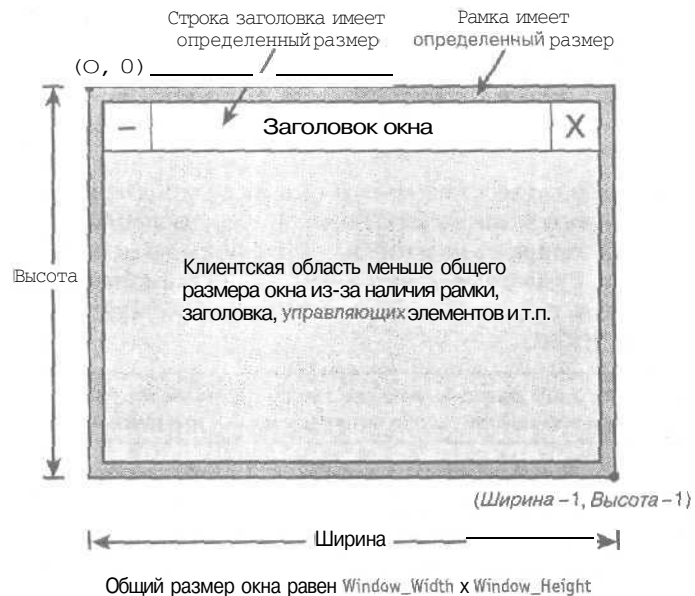


Рис. 3.13. Область окна и клиентская область

Для решения проблемы нам надо изменить размер окна таким образом, чтобы размер клиентской области в точности соответствовал переданным размерам. Вот код, решающий поставленную задачу:

```
// Изменение размеров окна
if (WINDOWED_APP)
{
    // Изменяем размер окна таким образом, чтобы
    // клиентская область имела запрошенный размер (учет
    // границ окна и управляющих элементов)
    RECT window_rect = {0, 0, WINDOW_WIDTH - 1,
                        WINDOW_HEIGHT - 1};

    // Вызов для изменения window_rect
    AdjustWindowRectEx(&window_rect,
        GetWindowStyle(main_window_handle),
        GetMenu(main_window_handle) != NULL,
        GetWindowExStyle(main_window_handle));

    // Сохранение глобальных переменных, необходимых для
    // работы DDraw_Flip()
    window_client_x0 = -window_rect.left;
    window_client_y0 = -window_rect.top;

    // Изменение размеров окна
    MoveWindow(main_window_handle,
        0, // Координата x
        0, // Координата y
        window_rect.right - window_rect.left, // Ширина
```

```

window_rect.bottom - window_rect.top, //Высота
FALSE);

// Вывод окна
ShowWindow(main_window_handle, SW_SHOW);
} // if windowed

```

Имеются и другие возможности добиться того же результата, но обычно я поступаю таким образом, как показано **выше**. Можно, например, выяснить, какие **управляющие** элементы имеет окно, запросить их размеры у **Windows**, а затем **вычислить**, каким должен быть размер окна с **управляющими** элементами. Каким бы образом вы ни поступали, главное, чтобы в результате размер клиентской области окна был равен **WINDOW\_WIDTH**×**WINDOW\_HEIGHT**.

#### НА ЗАМЕТКУ

Если **WINDOWED\_APP** равно 0, окно создается с флагом **WM\_POPUP** и не имеет никаких управляющих элементов, так что нет необходимости в изменении его размеров.

После того, как окно оказывается созданным, а его размеры (при необходимости) изменены, вызывается функция **Game\_Init()**. В эту функцию позже вы добавите всю необходимую инициализацию вашей игры. В настоящий момент эта функция выполняет необходимые в любом случае действия, в частности — инициализацию **DirectDraw**.

```

DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT,
           WINDOW_BPP, WINDOWED_APP);

```

Так что, как видите, вам **всего** лишь достаточно указать необходимые величины в некоторых макроопределениях и не **заботиться** об остальном.

#### НА ЗАМЕТКУ

Вы могли обратить внимание на вызовы **System Parameters Info()** вокруг главного цикла сообщений. Эти вызовы заставляют **Windows** думать, что включен режим сохранения экрана, и игнорировать нажатия <Alt+Tab>. Дело в том, что если вы не обрабатываете нажатие этих клавиш (а вы его не обрабатываете), приложение **DirectX** может работать некорректно. Вызовы **System Parameters Info()** позволяют вам забыть об этом вопросе. Если вас интересуют технические подробности — обратитесь к **DirectX SDK**. Вкратце — при потере приложением фокуса ввода и получении его обратно вы должны восстановить все утраченные поверхности, заново захватить устройства ввода и т.д. — словом, появляется масса причин для головной боли...

## Использование и компиляция консоли игры

Рассмотренный нами код в файле **T3DCONSOLE2.CPP** — всего лишь заготовка, шаблон, который вы используете для создания реального приложения. Вам просто надо поместить свой код в функции **Game\_\***(**)**. Однако для того, чтобы скомпилировать его, вам нужны следующие файлы;

- **T3DLIB1.CPP|H** — модуль работы с **DirectDraw**;
- **T3DLIB2.CPP|H** — модуль работы с **DirectInput**;
- **T3DLIB3.CPP|H** — модуль работы с **DirectSound** и **DirectMusic**.

Кроме того, в вашем каталоге должны быть файлы

- **PALDATA1|2.PAL** — палитры по умолчанию для 256-цветного режима

Кроме того, вы должны компоновать ваше приложение вместе с

- **DDRAW.LIB, DSOUND.LIB, DINPUT.LIB** и **DINPUT8.LIB**

Не забудьте установить в вашем компиляторе *опцию* создания *.EXE-приложения* Win32, добавить все необходимые библиотечные файлы DirectX в список компоновки, а также указать пути поиска компилятора таким образом, чтобы он мог найти все необходимые заголовочные файлы.

Просто интереса ради я скомпилировал консоль без каких-либо добавлений, получив файл T3DCONSOLE2.EXE. Это приложение ничего не *делает* — просто создает оконный дисплей 640x480x16. Однако для запуска этого приложения вы должны находиться в 16-битовом режиме.

**ВНИМАНИЕ**

Оконное приложение на *основе* консоли игры *не* изменяет *глубину* цвета рабочего стола. При желании вы можете изменить ее, но это чревато катастрофическими последствиями. Дело в том, что если вы запустили другое приложение, а затем *позволяете* приложению DirectDraw изменить глубину цвета экрана, то первое *приложение* может в результате функционировать некорректно и даже привести к краху системы при получении фокуса ввода. В полноэкранном приложении вы можете делать все, что хотите, но в оконном первым делом следует выяснить, с какой глубиной цвета вы имеете дело и соответствует ли она глубине цвета вашего приложения.

Я бы легко мог сделать приложение полноэкранным — для этого понадобилось бы просто внести маленькое изменение в одну строку.

```
tfdefine WINDOWED_APP 0 // 0 - полноэкранное  
// приложение, 1 - оконное
```

Однако оконное приложение выглядит более интересно, и его легче отлаживать — поэтому в книге мы будем иметь дело в основном с оконными приложениями. Однако, когда игра создана и отлажена — имеет смысл перевести ее в полноэкранный режим.

## Образцы приложений T3DLIB

Мы добрались почти до конца данной главы. Теперь для полноты изложения я просто хочу показать вам несколько примеров приложений, созданных на основе консоли игры. Эти приложения — уже не просто пустое окно; они выполняют реальные действия.

Далее в книге к уже *имеющимся* частям игрового процессора T3DLIB будут добавлены новые модули, а пока мы ограничимся только имеющимися в нашем распоряжении возможностями работы с DirectX.

### Оконные приложения

В качестве примера оконного приложения (файлы DEMO113\_1.CPP|EXE) я использовал преобразованную демонстрационную программу из первой книги. Копия экрана этой программы показана на рис. 3.14. Не забудьте — для того, чтобы эта программа корректно работала, ваш экран должен находиться в 16-битовом режиме!

Эта программа показывает не только работу в 16-битовом режиме, но и загрузку растровых изображений, использование объектов *блиттера* и звуковых эффектов. Право же, код этого приложения стоит того, чтобы внимательно его изучить!

### Полноэкранное приложение

Написание полноэкранного приложения благодаря наличию разработанной библиотеки ничем по сути не отличается от написания оконного приложения. Более того, такое приложение даже лучше в том плане, что при его работе не возникает проблем, связанных с *совмест-*

ным использованием ресурсов. Кроме того, в 256-цветном режиме в полноэкранном приложении вы имеете право распоряжаться всей палитрой полностью, в то время как в оконном режиме цвета в начале и конце палитры зарезервированы для использования Windows. Еще одним существенным достоинством полноэкранного режима работы является то, что вы можете переключать разрешение и цветовую глубину как вам вздумается, без оглядки на настройки рабочего стола.

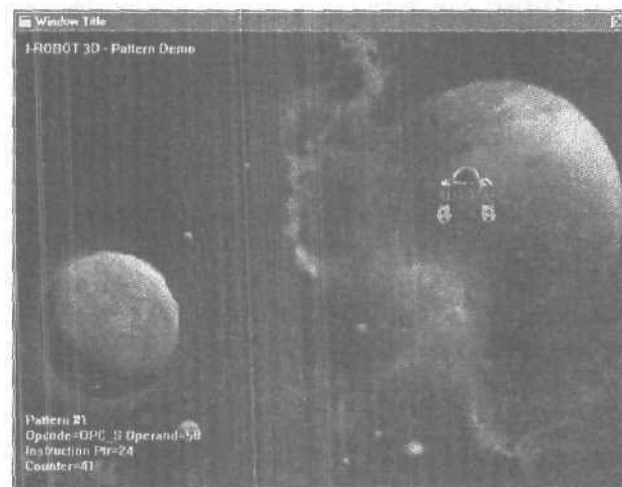


Рис. 3.14. Копия экрана оконного приложения, демонстрирующего работу искусственного интеллекта

В качестве основы для демонстрации полноэкранного режима (файлы DEMO113\_2.CPP|EXE) я также взял программу из первой книги. Копия экрана программы приведена на рис. 3.15. В этой программе также использованы загрузка изображений, объекты блиттера, а также 256-цветная палитра.

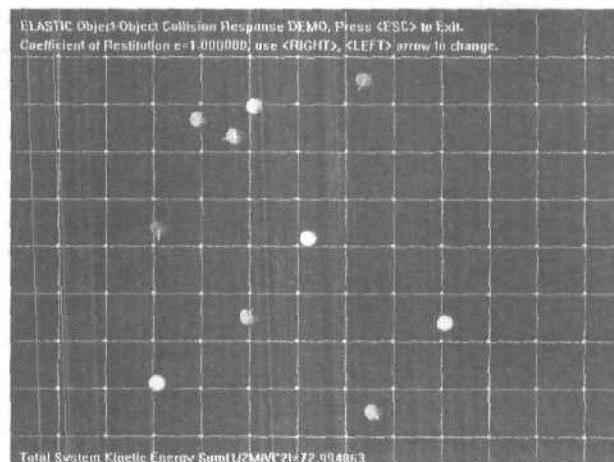


Рис. 3.15. Полноэкранная демонстрационная программа с 8-битовой глубиной цвета

## Звук и музыка

Работа со звуком и музыкой под управлением `DirectSound` и `DirectMusic` не столь сложна, но и простой ее не назовешь. Однако модуль `T3DLIB2.CPP` делает использование звука и музыки очень простым, в чем вы можете сами убедиться, *познакомившись* с демонстрационным приложением `DEMO113_3.CPP|EXE`, копия экрана которого показана на рис. 3.16. Это простое приложение Windows с меню, при помощи которого вы можете выбрать для воспроизведения *MIDI-мелодию*, на фоне которой воспроизвести те или иные звуковые эффекты (также выбираемые при помощи меню).

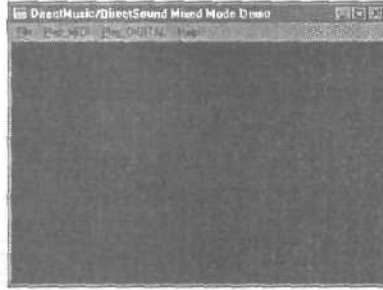


Рис. 3.16. Демонстрационная программа воспроизведения музыки и звуков

Данная программа основана на несколько урезанном шаблоне консоли *игры* (поскольку нас не интересует графика), зато она использует разные ресурсы, такие как меню и курсор. Соответственно, вы должны добавить в проект следующие файлы:

- `DEMO113_3.RC` — файл ресурсов Windows с меню;
- `DEMO113_3.RES.H` — заголовочный файл с идентификаторами ресурсов;
- `T3DX.ICO` — пиктограмма курсора, используемого в программе.

Эта программа — обычное приложение `DirectX`, так что вы должны включить в проект библиотечные файлы `DirectX`. Технически для *компиляции* данной программы библиотека `DDRAW.LIB` не нужна, но пусть она останется в списке компонентных файлов — для безопасности. Поскольку в программе не используется никакая *функциональность* из модулей `T3DLIB1.CPP` и `T3DLIB2.CPP`, включать их в проект не надо.

## Работа с устройствами ввода

Последние примеры демонстрируют работу с разными устройствами ввода. Как вы знаете, в играх в первую очередь используются три *следующие* устройства ввода:

- клавиатура;
- мышь;
- джойстик.

Сегодня понятие "джойстик" объединяет множество разных устройств — *обычно* все устройства, не являющиеся мышью или клавиатурой, классифицируются как джойстик.

Использование библиотеки для работы с устройствами ввода очень простое. Все функции библиотеки содержатся в модуле `T3DLIB2.CPP` и поддерживают работу с клавиатурой, мышью и джойстиком. Всего лишь несколькими вызовами функций вы

можете инициализировать DirectInput, захватить устройство ввода и считать с него информацию в основном цикле событий. На прилагаемом компакт-диске имеются демонстрационные приложения для каждого из перечисленных устройств ввода. Конечно, в своих программах вы можете использовать все их одновременно — в демонстрационных программах они используются по отдельности исключительно в методических целях.

## Клавиатура

Как и прочие демонстрационные программы, пример приложения, работающего с клавиатурой, основан на коде из предыдущей книги. Это программа, находящаяся на прилагаемом компакт-диске под именем DEMO13\_4.CPP|EXE, копия экрана которой показана на рис. 3.17. Здесь при помощи клавиатуры вы можете управлять перемещением объекта **блиттера** по экрану. Если вы обратитесь к коду данной программы, то увидите, что чтение клавиатуры сводится к заполнению массива из 256 байтов, каждый из которых представляет одну клавишу.

UCHAR keyboard\_state[256]; // Таблица состояния клавиатуры

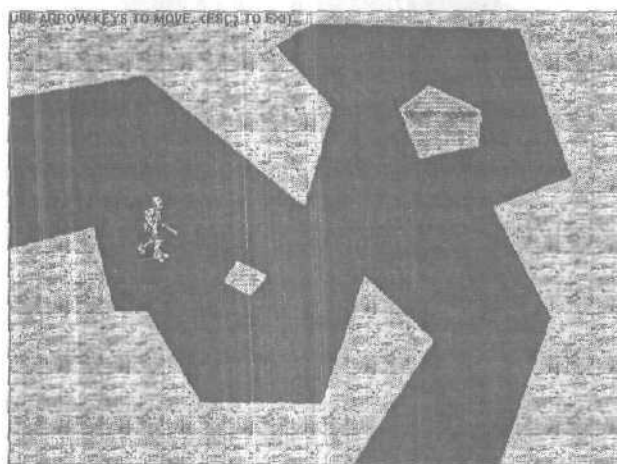


Рис. 3.17. Демонстрационная программа использования клавиатуры

Для доступа к элементам этой таблицы используются константы `DIK_*` DirectInput, перечисленные в табл. 3.2. Все, что вы должны сделать, — это проверить, нажата ли интересующая вас клавиша. После того как вы получили информацию о состоянии клавиатуры при помощи вызова `DInput_Read_Keyboard()`, вы можете воспользоваться проверкой наподобие следующей.

```
if (keyboard_state[DIK_UP])
{
    // Код, выполняющийся при нажатой клавише "↑"
} // if
```

Для компиляции демонстрационной программы вам нужны все библиотечные файлы DirectX и модули библиотеки `T3DLIB`.

# Мышь

Пример приложения, работающего с мышью, также основан на коде из предыдущей книги. Это программа, находящаяся на прилагаемом компакт-диске под именем DEMO113\_5.CPP[EXE, копия экрана которой показана на рис. 3.18. В программе использован ряд интересных приемов, таких как выбор объекта, отслеживание мыши и т.п. — так что эта программа настоятельно рекомендуется к серьезному самостоятельному изучению.

Мышь может работать в двух режимах — абсолютном и относительном. Я предпочитаю относительный режим работы, поскольку я всегда знаю, где я нахожусь. Соответственно, и весь API разработан мною для работы в относительном режиме. Всякий раз, когда вы считываете *информацию* о мыши, вы получаете ее смещение от предыдущего положения. Считывание информации производится функцией `DInput_Read_Mouse()`, после чего полученные данные помещаются в глобальную переменную

```
DIMOUSESTATE mouse_state; // Состояние мыши
```

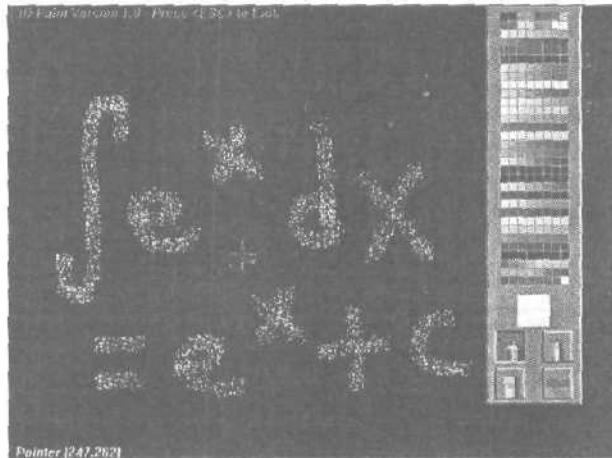


Рис. 3.18. Демонстрационная программа использования мыши

Напомню, что структура DIMOUSESTATE выглядит следующим образом.

```
typedef struct DIMOUSESTATE{
    LONG lX;      // Ось x
    LONG lY;      // Ось y
    LONG lZ;      // Ось z
    BYTE rgbButtons[4]; // Состояние кнопок
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Для компиляции демонстрационной программы вам нужны все библиотечные файлы DirectX и модули библиотеки T3DLIB.

## Джойстик

В настоящее время "джойстиком" может быть все, что угодно — от игрового пульта до рулевого колеса. Однако любое такое устройство представляет собой не более чем набор двух типов устройств ввода:

- аналоговых осей;
- КНОПОК.

Конечно, в `DirectInput` имеется ряд констант для определения наиболее часто встречающихся типов джойстиков, но в общем все они используются совершенно одинаково — вы получаете от устройства ряд переменных значений от аналоговых ползунков или подобных им устройств, и логические значения, представляющие состояние кнопок. Данные, считанные функцией `DInput_Read_Joystick()`, вносятся в глобальную переменную

`DIJOYSTATE joy_state; // Состояние джойстика`

тип которой —

```
typedef struct DIJOYSTATE {
    LONG lX;      // Ось x
    LONG lY;      // Ось y
    LONG lZ;      // Ось z
    LONG lRx;     // Поворот вокруг оси x
    LONG lRy;     // Поворот вокруг оси y
    LONG lRz;     // Поворот вокруг оси z
    LONG rgSlider[2]; // Положения ползунка u,v
    DWORD rgdwPOV[4]; // Индикаторы обзора
    BYTE rgbButtons[32]; // Состояние кнопок 0..31
} DIJOYSTATE, *LPDIJOYSTATE;
```

Пример приложения, работающего с джойстиком, представляет собой переработанный код одной из демонстрационных программ из предыдущей книги. Это программа, находящаяся на прилагаемом компакт-диске под именем `DEMO113_6.CPP|EXE`, копия экрана на которой показана на рис. 3.19. Для компиляции демонстрационной программы вам нужны все библиотечные файлы `DirectX` и модули библиотеки `T3DLIB`.

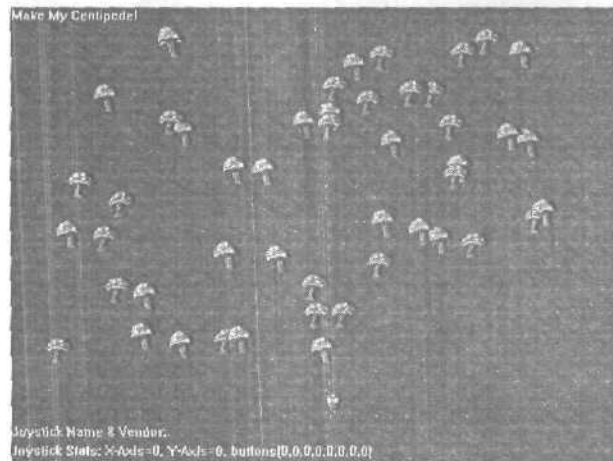


Рис. 3.19. Демонстрационная программа использования джойстика

## Резюме

Эта глава по сути представляет собой переход от книги *Программирование игр для Windows. Советы профессионала* к данной книге. Вы должны понять, что трехмерная графика не имеет никакого отношения к `DirectX` или `Windows`. Это математика, ма-

тематика и **еще** раз математика. Все, что нам надо для экспериментов с трехмерной графикой, — это несколько **функций** и набор буферов кадров.

После знакомства с данной главой (даже если вы не читали **предыдущую** книгу) вы должны получить знания об используемом в книге API, достаточные для того, чтобы в дальнейшем при работе со сложными математическими вопросами не отвлекаться на такие мелочи, как графический вывод или реализация музыкального сопровождения. Все это реализовано в библиотеке **T3DLIB** при помощи удивительно малого количества **функций**.



# ЧАСТЬ II

## Трехмерная математика и преобразования

### В этой части...

#### Глава 4

Запутанный мир математики 221

#### Глава 5

Создание математической библиотеки 309

#### Глава 6

Введение в трехмерную графику 409

#### Глава 7

Визуализация трехмерных каркасных объектов 515



# ГЛАВА 4

## Запутанный мир математики

### В этой главе...

• Математические обозначения	222
• Двумерные системы координат	223
• Трёхмерные системы координат	228
• Тригонометрия	234
• Векторы	238
• Матрицы и линейная алгебра	247
• Обращение матриц и решение систем линейных уравнений	252
• Фундаментальные геометрические объекты	263
• Использование параметрических уравнений	271
• Введение в кватернионы	278
• Дифференциальное исчисление	291

Для того чтобы создавать компьютерные игры, просто необходимы знания алгебры и тригонометрии. Взгляды на этот вопрос в последние годы определенно изменились. Припоминаю, как несколько лет назад я пытался уговорить редактора *Game Developer Magazine* поручить мне написать статью о векторах, и как он ответил мне, что это слишком сложная математика. Сегодня я беру в руки этот журнал и нахожу в нем массу векторных вычислений! Исходя из этого положения вещей, данная глава призвана освежить в памяти читателя основы математики, необходимые для трёхмерной компьютерной графики.

Я не могу обучить в одной главе тому, чему обучают серьезные преподаватели в течение нескольких лет, но я и не ставлю перед собой такую цель — я **только** хочу "встряхнуть" вашу память, а если этого окажется недостаточно — что ж, вам придется засесть за учебники самостоятельно. Конечно, если вы хорошо знаете математику, вы можете пропустить эту главу, в которой начинается построение математической библиотеки для разработки трехмерных игр. Но, пропустите вы ее или нет, изложенный в ней материал останется **неизменным**:

- математические обозначения;
- системы координат;
- тригонометрия;
- векторы;
- матрицы и линейная алгебра;
- основы геометрии;
- использование параметрических уравнений;
- введение в кватернионы;
- фундаментальные вычисления.

## Математические обозначения

Математика подобна любому языку. Она использует набор символов, которые представляют математические концепции, формулы, уравнения и операции. Одна из сложностей высшей математики состоит в том, что в ней слишком много греческих букв! Так что для начала я хочу познакомить вас с часто используемыми символами и соглашениями, которых я буду придерживаться далее в этой книге. Хотя в настоящий момент некоторые из перечисленных концепций могут показаться вам непонятными, дальше все станет на свои места, а пока смотрите на табл. 4.1 просто как на карту определений, с которыми вам придется встретиться немного позже. Ведь карта нужна в начале пути, а не после того, как весь путь пройден.

**Таблица 4.1. Типы математических объектов**

Тип	Обозначение	Пример
Скаляр	Строчные буквы	$a, b, x, y$
Бесконечность	$\infty$	$-\infty, +\infty$
Угол	Строчные греческие буквы	$\theta, \phi, \alpha, \beta$
Вектор	Полужирные строчные буквы	$\mathbf{u}, \mathbf{v}, \mathbf{a}_x, \mathbf{a}_y$
Кватернион	Полужирные строчные буквы	$\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}$
Элементы вектора	$\langle$ Компоненты $\rangle$	$\langle ux, uy, uz \rangle$
Вектор-строка	$[$ Компоненты $]$	$[ix, uy, uz]$
Точка	Полужирные строчные буквы	$\mathbf{a}, \mathbf{p}_1, \mathbf{p}_2$
Матрица	Полужирные прописные буквы	$\mathbf{A}, \mathbf{B}, \mathbf{M}_2$
Треугольник	Символ $\Delta$ , за которым следуют точки	$\Delta \mathbf{p}_0 \mathbf{p}_1 \mathbf{p}_2, \Delta abc$

*Примечание. Все типы могут иметь верхние или нижние индексы.*

Теперь, когда мы познакомились с типами математических объектов, рассмотрим некоторые стандартные математические операторы, перечисленные в табл. 4.2.

**Таблица 4.2. Математические операторы**

Операция	Обозначение	Пример
Скалярное умножение <sup>1</sup>	$x, \cdot$	$3 \times 5, a \cdot b$
Матричное умножение	$\times$	$A \times B$
Скалярное произведение		$u \cdot v$
Суммирование	$\sum \text{выражение}$	
Абсолютное значение	$ \text{выражение} $	$ -35 $
Длина	$ \text{выражение} $	$ u $
Детерминант (определитель)	$\det(\text{выражение})$	$\det(M)$
Транспонирование	Верхний индекс $t$	$v^t$
Интегрирование	$\int \text{выражение}$	
Дифференцирование	Символ производной $'$	$x', x^*$

Разумеется, могут использоваться и другие, дополнительные обозначения, но в таблицах приведены наиболее употребительные, стандартные обозначения, признанные в большинстве математических книг.

**ВНИМАНИЕ**

В математической литературе зачастую для обозначения детерминанта матрицы используется обозначение  $|M|$ , а для длины или нормы вектора — запись  $\|v\|$ .

## Двумерные системы координат

Познакомившись с используемыми математическими обозначениями, приступим к рассмотрению двумерных систем координат, а затем перейдем к более сложным трехмерным системам координат.

### Двумерные декартовы координаты

Наиболее распространенная двумерная система координат, с которой приходилось иметь дело каждому из нас — это *декартовы координаты*. Декартовы координаты основаны на паре взаимно перпендикулярных осей  $x$  и  $y$ , как показано на рис. 4.1. Положительное направление оси  $x$  — вправо (естественно, *отрицательное направление* оси  $x$  — влево). Положительное и отрицательное направления оси  $y$  — соответственно, вверх и вниз. Точка, где координаты  $x$  и  $y$  равны 0, называется *началом координат*. Если воспользоваться техническими терминами, то ось  $x$  называется *осью абсцисс*, а ось  $y$  — *осью ординат*.

<sup>1</sup> В данной главе символ умножения “ $\cdot$ ” явно указывается во всех формулах, однако в дальнейшем в книге он может быть опущен там, где его наличие очевидно из контекста.

Кроме того, на рисунке вы видите четыре квадранта, которые разделяют плоскость на четыре части в **соответствии** со знаками координат  $x$  и  $y$ . В табл. 4.3 показаны знаки координат для каждого из квадрантов.

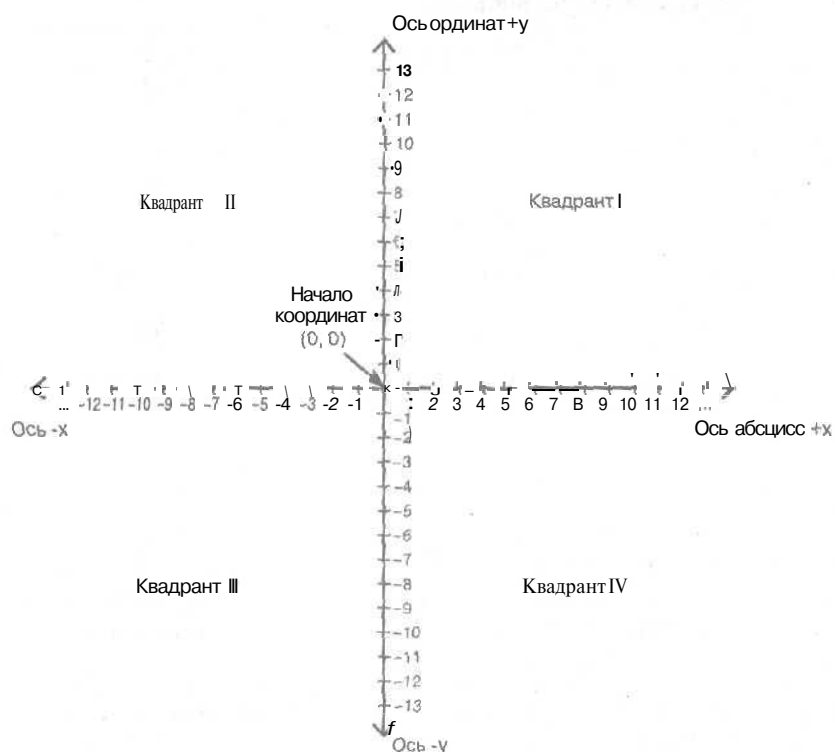


Рис. 4.1. Декартова система координат

Таблица 4.3. Знаки координат в квадрантах

Квадрант	Знак $x$	Знак $y$
I	+	+
II	-	+
III	-	-
IV	+	-

**СОВЕТ**

**Многие** графические алгоритмы оптимизированы для работы в первом квадранте, а затем решения для других квадрантов (октантов в трехмерном случае) получаются в результате использования преобразований симметрии.

Наконец, для того, чтобы указать положение любой точки в двумерной декартовой системе координат, нам необходимо указать  $x$  и  $y$ -компоненты координат. Например,  $p(5,3)$  обозначает, что значение координаты  $x$  данной точки равно 5, а значение координаты  $y$  — 3, как показано на рис. 4.2. Все слишком просто для вас? Потерпите немного...

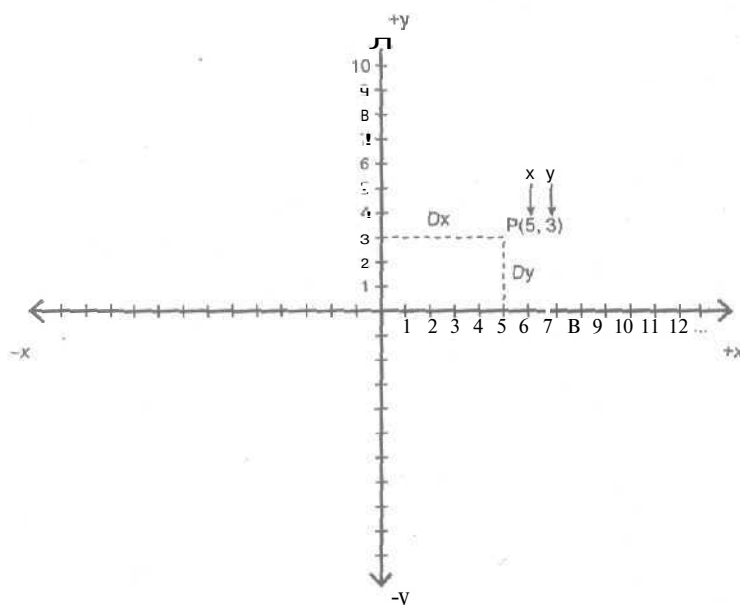


Рис. 4.2. Положение точки в декартовой системе координат

## Двумерные полярные координаты

Следующий вид системы координат, *поддерживающей* две степени свободы, — *полярные координаты*. Полярные координаты используются, например, в игре *Wolfenstein* и в технологии расчета лучей. Полярные координаты основаны на том, что положение точки на плоскости можно определить не только двумя координатами  $(x, y)$ , но и направлением и расстоянием от начала координат (на рис. 4.3 показана стандартная полярная система координат). Как видите, для указания положения точки используются две *переменные*: расстояние  $r$  от начала координат, или *полюса*, и направлением, или углом  $\theta$ . Таким образом, запись  $p(r, \theta)$  означает, что точка  $p$  расположена под углом  $\theta$  относительно начальной оси (обычно это  $x$ -ось), измеряемым в направлении против часовой стрелки, и на расстоянии  $r$  в данном направлении. На рис. 4.4 показаны примеры расположения точек  $p_1(10, 30^\circ)$  и  $p_2(6, 150^\circ)$  в полярной и декартовой системах координат.

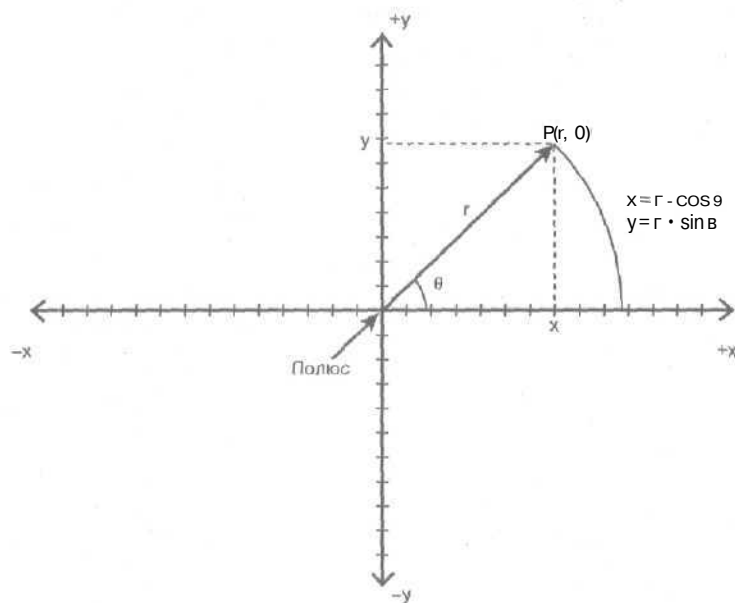


Рис. 4.3. Двумерная полярная система координат

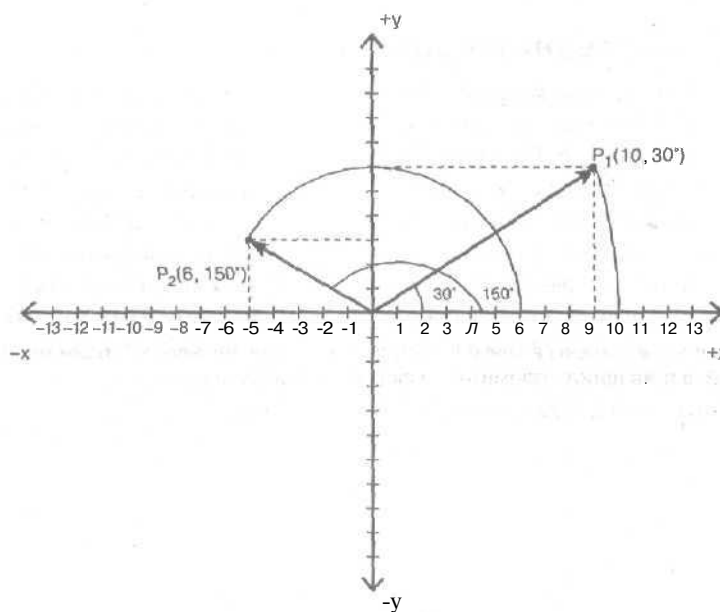


Рис. 4.4. Пример расположения точек в двумерной полярной системе координат

## Преобразования между полярными и декартовыми координатами

Случается (и не так редко), что нужно преобразовать полярные координаты некоторой точки в декартовы (или наоборот). Как выполнить такое преобразование? Это очень просто — надо лишь немного вспомнить тригонометрию. Взгляните на рис. 4.5, на кото-

ром показан обычный треугольник в первом квадранте декартовой системы координат. Если рассматривать гипотенузу треугольника как отрезок от начала координат до интересующей нас точки P, то мы получим следующие формулы для преобразования полярных координат в декартовы.

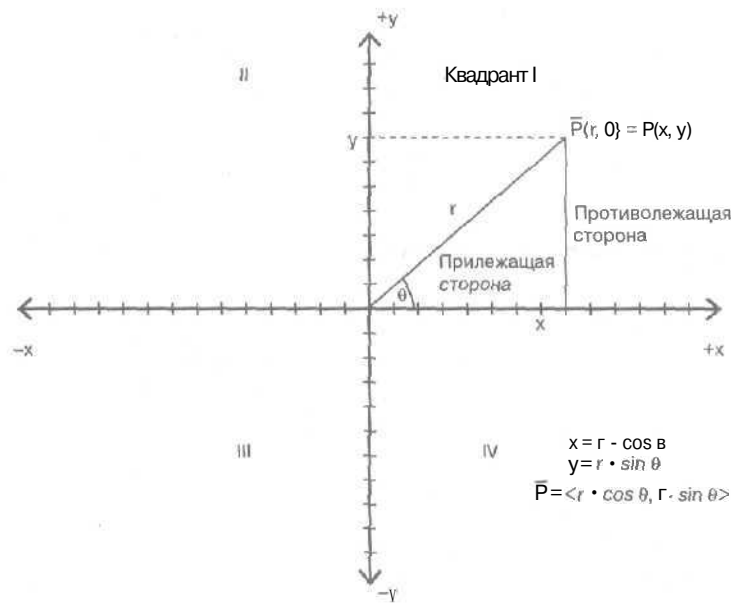


Рис. 4.5. Геометрическая интерпретация преобразования между полярными и декартовыми координатами

#### Уравнение 4.1. Преобразование полярных координат $p(r, \theta)$ в декартовы $p(x, y)$

$$x = r \cdot \cos \theta$$

$$y = r \cdot \sin \theta$$

##### СОВЕТ

Если вы не сильны в тригонометрии, потерпите **немного** — позже мы расскажем и о ней, **а пока** просто примите эти формулы на **веру**.

Преобразование из декартовых координат в полярные немного хитрее. Проблема в том, что нам нужен угол, который гипотенуза, проведенная из начала координат в точку  $p(x, y)$ , образует с осью  $x$ , и ее длина. И здесь нам на **помощь** вновь приходит тригонометрия. Для поиска угла мы можем воспользоваться тангенсом, а значение  $r$  определяется из теоремы Пифагора.

#### Уравнение 4.2. Преобразование декартовых координат $p(x, y)$ в полярные $p(r, \theta)$

$$\text{Дано: } x^2 + y^2 = r^2,$$

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctg(y/x)$$

Применим приведенные формулы к точке  $(3, 4)$ , располагающейся, как видно из рис. 4.6, в первом квадранте.

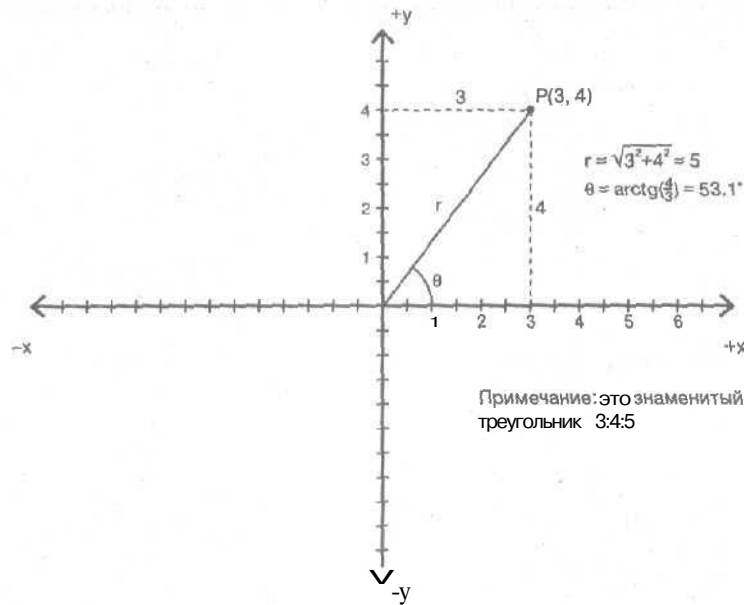


Рис. 4.6. Пример преобразования декартовых координат в полярные

Подставляя значения  $x = 3$ ,  $y = 4$  в уравнение 4.2, получаем:

$$r = \sqrt{3^2 + 4^2} = 5$$

$$\theta = \arctg(4/3) = 53.1^\circ$$

Если посмотреть на рис. 4.6, то корректность вычислений становится очевидной.

#### СОВЕТ

Если внимательно посмотреть на уравнение 4.2, то можно заметить проблему оси ординат ( $x = 0$ ). Другими словами, при углах  $0 = 90^\circ$  и  $270^\circ$  значение тангенса не определено (точнее, оно равно бесконечности). Таким образом, используя уравнение 4.2, необходимо дополнительно проверять выполнение условия  $x = 0$  (как вы знаете, угол при этом равен  $90^\circ$ ).

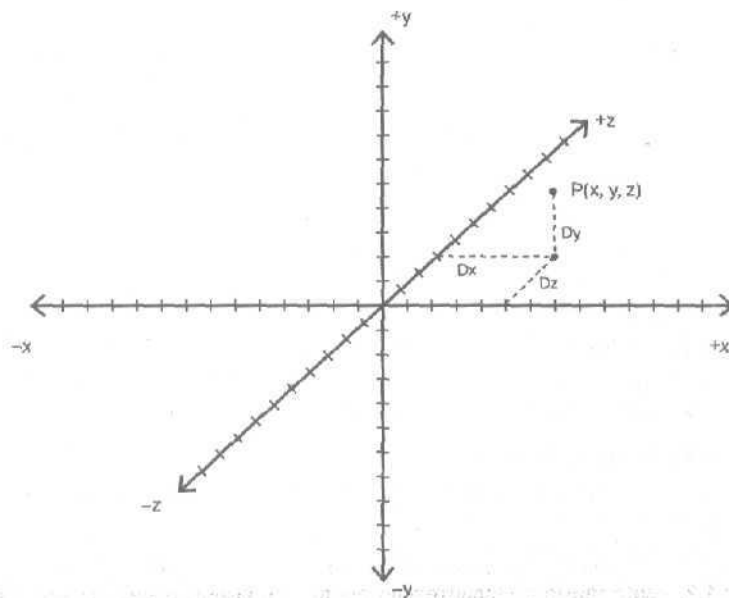
В заключение **следует** сказать, что полярная система координат часто оказывается очень полезной, так что понимание того, как преобразовывать декартовы координаты в полярные и обратно, очень важно для успешного решения множества задач, в том числе задач слежения и навигации.

## Трехмерные системы координат

Теперь, когда вы познакомились с двумерными системами координат, давайте перейдем к трехмерным системам координат. Если вы уже занимались программированием игр, вероятно, вы уже хорошо знакомы с декартовыми трехмерным координатами. Однако я хочу познакомить вас с цилиндрическими и сферическими системами координат, которые позже помогут нам при отображении текстур и работе над рядом других специальных эффектов. Но начнем мы все же с декартовых трехмерных координат.

## Трехмерные декартовы координаты

Трехмерные декартовы координаты **идентичны** двумерным координатам, но к ним добавлена третья  $z$ -ось. Таким образом мы получаем систему с тремя степенями свободы, основанную на трех взаимно *ортогональных* (перпендикулярных) **осях**. Следовательно, чтобы определить местоположение точки  $p$  в трехмерном пространстве, нам нужны три координаты:  $x$ ,  $y$  и  $z$ , или, в компактной записи,  $p(x, y, z)$ . Кроме того, три оси координат образуют три плоскости:  $x$ - $y$ ,  $x$ - $z$  и  $y$ - $z$  (рис. 4.7). Эти плоскости очень важны; каждая из них делит пространство на два полупространства. Это очень важная концепция, с которой мы встретимся во множестве алгоритмов. А теперь — микрозадача: у нас есть три плоскости, каждая из которых делит пространство на два **полупространства**. Сколько всего различных подпространств образуется в такой трехосной системе? Ответ — **восемь**! Итак, в трехмерной системе у нас имеется восемь октантов (рис. 4.8).



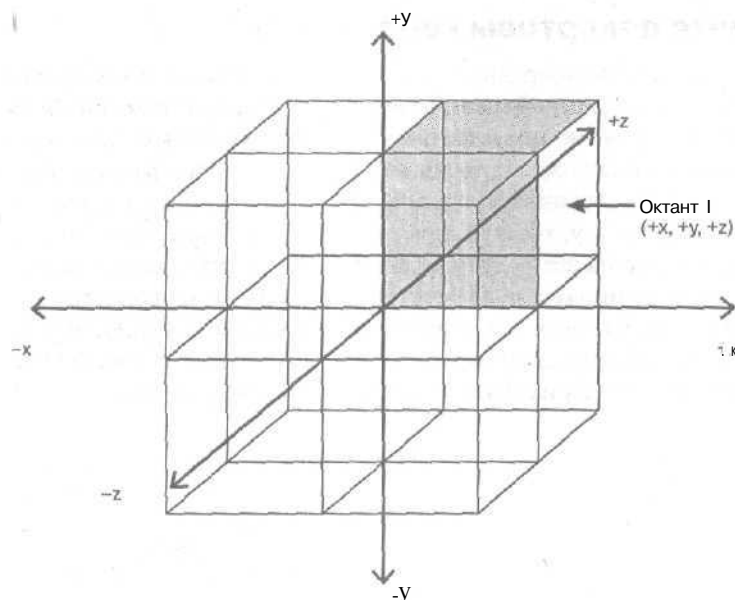
Левая система координат

Рис. 4.7. Трехмерная декартова система координат

Однако в такой системе тоже не без проблем — поскольку у оси  $z$  есть два возможных направления (**ориентации**), возможны две различные трехмерные декартовы системы координат — *левая* и *правая*.

### СОВЕТ

Чтобы **понять**, откуда произошли эти названия, попробуйте расположить большой, средний и указательный пальцы руки наподобие осей координат. Если ось  $x$  соответствует большому пальцу, а  $y$  — среднему, то указательный палец указывает направление оси  $z$  в соответствующей системе координат: если это *левая* рука — то в левой, а если правая — то в правой. Еще один вариант — раскройте ладонь и оставьте большой палец. Если большой палец направлен вдоль оси  $x$ , *воображаемая* ось  $y$  выходит из ладони, то пальцы указывают направление оси  $z$  в соответствующей системе координат.



Левая система координат

Рис. 4.8. Октанты в трехмерной системе координат

### Левая система координат

Левая система координат (left-handed system, **LHS**) показана на рис. 4.9. В этой системе координат, если оси  $x$  и  $y$  представляют собой горизонтальную и вертикальную оси на экране, то положительное направление оси  $z$  идет вглубь экрана.

### Правая система координат

Правая система координат (right-handed system, **RHS**) показана на рис. 4.10. В этой системе координат, если оси  $x$  и  $y$  представляют собой горизонтальную и вертикальную оси на экране, то отрицательное направление оси  $z$  идет вглубь экрана, а положительное, соответственно, от экрана к наблюдателю.

#### НА ЗАМЕТКУ

На самом деле принципиальных различий между этими системами координат нет, но при рассмотрении трехмерных виртуальных машин чаще будет использоваться левая система координат, хотя для лучшего понимания местами может использоваться и правая.

В основном в этой книге будет использоваться именно декартова трехмерная система координат, но иногда оказывается проще решать задачу в терминах углов и направлений — например, при работе с трехмерными камерами. Поэтому не помешает так же хорошо понять и устройство двух других трехмерных систем координат.

### Трехмерные цилиндрические координаты

Появление дополнительной степени свободы по сравнению с двумерными координатами позволяет из двумерной полярной системы координат получить две трехмерные — цилиндрическую и сферическую. Ясно, что обе могут быть отображены одна в другую,

и обе находят свое применение. Ближе всего к двумерной полярной системе координат трехмерная *цилиндрическая система координат*, поскольку это по сути полярная система координат, к которой прибавлена третья координата  $z$ .

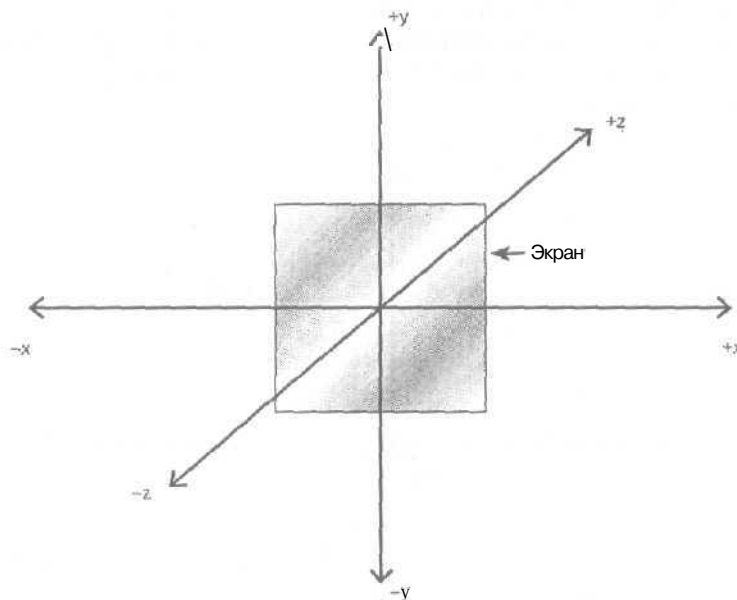


Рис. 4.9. Левая трехмерная система координат

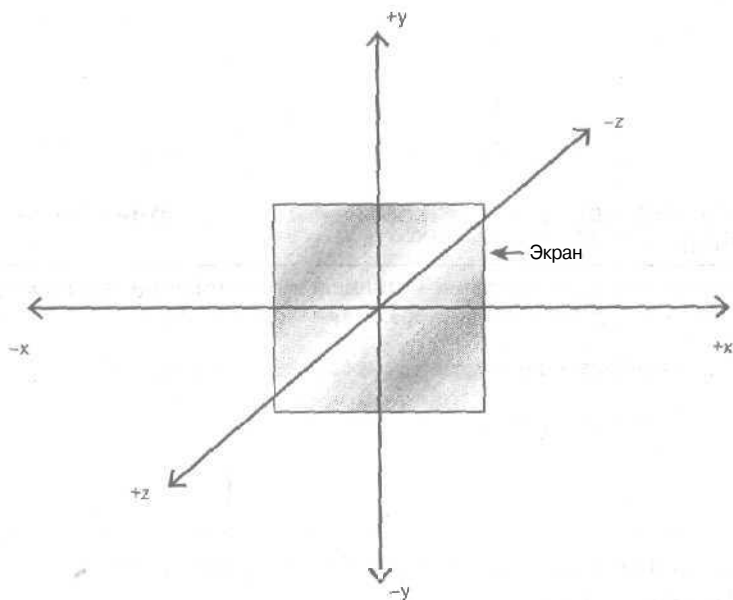


Рис. 4.10. Правая трехмерная система координат

Стандартная **цилиндрическая** система координат показана на рис. 4.11. Вы можете увидеть, что плоскость  $x$ - $y$  при  $z=0$  образует стандартную двумерную полярную систему координат, а **цилиндрические** координаты определяют положение **точки**, сперва указывая ее положение  $p(r, \theta)$  в двумерной системе координат, а затем "поднимая" ее вдоль оси  $z$  на необходимую высоту, **являющуюся** третьей координатой точки. Обратите внимание, что здесь использована правая система координат, и она повернута иначе, чем ранее — для лучшей наглядности.

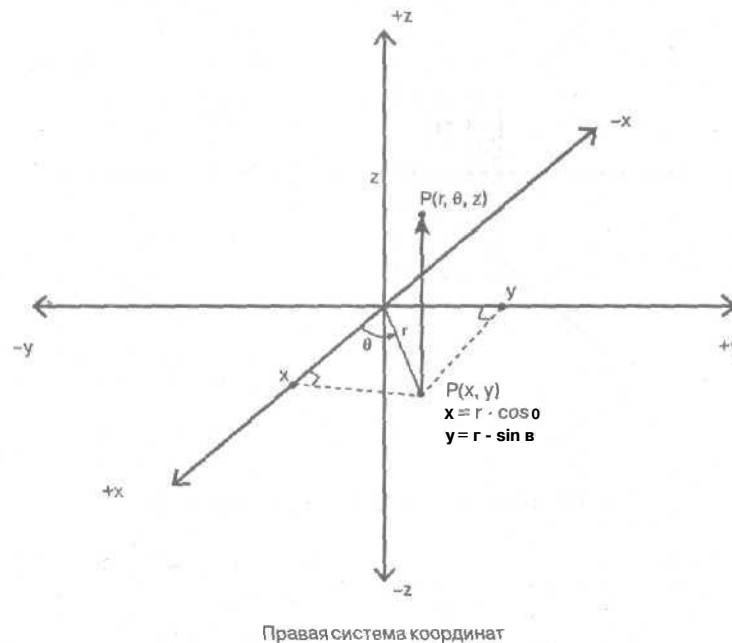


Рис. 4.11. Цилиндрическая система координат

### Преобразование между декартовыми и цилиндрическими координатами

Преобразование между декартовыми и цилиндрическими координатами тривиально: достаточно использовать двумерное преобразование и положить  $z = z$ .

**Уравнение 4.3. Преобразование цилиндрических координат  $p(r, \theta, z)$  в декартовы  $p(x, y, z)$**

$$x = r \cdot \cos \theta$$

$$y = r \cdot \sin \theta$$

$$r = z$$

При преобразовании декартовых координат в полярные мы вновь используем двумерное преобразование и полагаем  $z = z$ .

**ВНИМАНИЕ**

Здесь, как и в двумерных координатах, имеется проблема, связанная с углами  $\theta = 90^\circ$  и  $\theta = 270^\circ$ .

#### Уравнение 4.4. Преобразование декартовых координат $p(x, y, z)$ в цилиндрические $p(r, e, z)$

Дано:  $x^2 + y^2 = r^2$

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctg(y/x)$$

$$z = z$$

Цилиндрические координаты удобны для решения ряда задач, например, таких как управление камерой в "стрелялках" для отображения среды.

### Трехмерные сферические координаты

Трехмерные сферические координаты немного сложнее других систем координат. В них положение точки определяется двумя углами и расстоянием от начала системы координат (рис. 4.12). Соответственно, координаты точки записываются как  $p(\rho, \phi, \theta)$ , где  $\rho$  — расстояние от начала координат до точки  $p$ ,  $\phi$  — угол, который образует с положительным направлением оси  $z$  отрезок от начала координат к точке  $p$  (здесь используется правая система координат), и  $\theta$  — угол, образуемый проекцией отрезка от начала координат до точки  $p$  на плоскость  $xy$  (так же, как и в случае двумерных полярных координат; диапазон допустимых значений  $0 \leq \theta < 2\pi$ ).

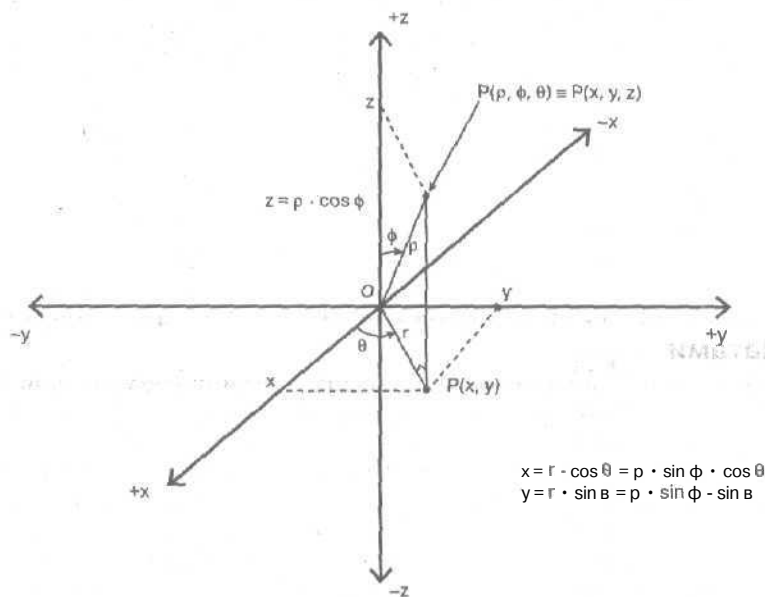


Рис. 4.12. Сферическая система координат

При выводе сферических координат мы вновь использовали свои знания о двумерных полярных координатах. Рассмотрим теперь преобразование координат из сферических в декартовы и обратно.

Сначала рассмотрим внимательнее рис. 4.12, поскольку описать решение при помощи одного только текста весьма затруднительно. Для преобразования сферических координат в

декартовы мы можем использовать **двухшаговый** процесс. Сначала мы проецируем отрезок от начала координат к интересующей нас точке на плоскость  $x-y$ , после чего задача сводится к уже решенной двумерной. Затем мы возвращаемся к поиску  $z$  через  $(\rho, \phi, \theta)$ .

---

**Уравнение 4.5. Преобразование трехмерных сферических координат  $\rho(\rho, \phi, \theta)$  в декартовы  $\rho(x, y, z)$**

---

Из проекции отрезка между началом координат и точкой получаем

$$r = \rho \cdot \sin \phi$$

$$z = \rho \cdot \cos \phi$$

Рассматривая проекцию на плоскости  $x-y$ , получим:

$$x = r \cdot \cos \theta$$

$$y = r \cdot \sin \theta$$

Подстановкой  $r$  в формулы для  $x$ ,  $y$  получаем окончательное решение:

$$x = \rho \cdot \sin \phi \cdot \cos \theta$$

$$y = \rho \cdot \sin \phi \cdot \sin \theta$$

$$z = \rho \cdot \cos \phi$$


---

**Уравнение 4.6. Преобразование трехмерных декартовых координат  $\rho(x, y, z)$  в сферические  $\rho(\rho, \phi, \theta)$**

---

Дано:  $x^2 + y^2 + z^2 = \rho^2$  и, аналогично,  $x^2 + y^2 = r^2$ . Тогда

$$r = \sqrt{x^2 + y^2},$$

$$\rho = \sqrt{x^2 + y^2 + z^2},$$

$$\theta = \arctg(y/x).$$

Значение  $\phi$  можно найти из соотношения  $r = \rho \cdot \sin \phi$ , откуда

$$\phi = \arcsin(r/\rho).$$

Можно также воспользоваться соотношением  $z = \rho \cdot \cos \phi$ , тогда

$$\phi = \arccos(z/\rho).$$


---

## Тригонометрия

Основная часть тригонометрии основана на анализе *прямоугольного треугольника* (рис. 4.13). Прямоугольный треугольник, как и любой другой, имеет три внутренних угла и три стороны. За точку отсчета обычно берется один из острых углов, который мы будем считать базовым. Сторона, противоположная базовому углу, называется *противолежащей*, сторона, соединяющая базовый угол с прямым — *прилежащей*, а сторона, соединяющая эти две — *гипотенузой*. Хотя все это звучит тривиально, постарайтесь не впасть в ошибку недооценки важности *треугольников* — я видел не так уж много графических алгоритмов, где бы так или иначе не использовались треугольники. Поэтому вы должны знать треугольники даже лучше своих пяти пальцев.

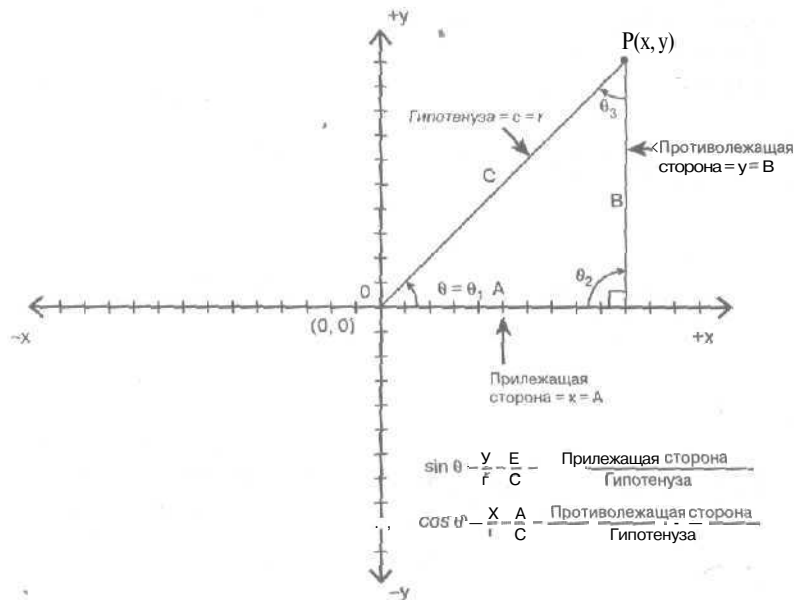


Рис. 4.13. Прямоугольный треугольник

## Прямоугольный треугольник

Вспомним основные факты, относящиеся к треугольникам.

НА ЗАМЕТКУ

Далее в книге встречаются как обозначение  $\pi$ , так и PI — в принципе, они взаимозаменяемы, разница лишь в том, что PI — это численная константа в вычислениях, являющаяся приближением числа  $\pi$ .

Таблица 4.4. Радианы и градусы

Угол в градусах	Угол в радианах
360°	2 $\pi$
180°	$\pi$
57.296°	$360^\circ/2\pi = 1$
1.0°	$2\pi/360 = 0.0175$

Факт 1. Полная окружность составляет 360°, или 2 $\pi$  радиан. Запомните — компьютерные функции  $\sin()$  и  $\cos()$  **работают с радианами, а не с градусами!** Некоторые часто встречающиеся значения показаны в табл. 4.4.

Факт 2. Сумма внутренних углов треугольника  $\theta_1 + \theta_2 + \theta_3 = \pi$  радиан (или 180°).

Факт 3. Обращаясь еще раз к рис. 4.13: сторона прямоугольного треугольника, противоположная углу  $\theta_1$ , называется **противоположащей**, ниже ее — **прилежащей** (обе они являются **катетами**), а длинная сторона называется **гипотенузой**.

Факт 4. Сумма квадратов длин катетов равна квадрату длины гипотенузы. Этот факт называется **теоремой Пифагора**. В обозначениях рис. 4.13 можно записать:  $A^2 + B^2 = C^2$ . Таким образом, зная две стороны прямоугольного треугольника, всегда можно найти третью,

Факт 5. Есть три основные тригонометрические функции — синус, косинус и тангенс. Их определения представлены ниже.

$$\cos \theta = \frac{\text{Прилежащая сторона}}{\text{Гипотенуза}} = \frac{x}{r} \quad (\text{область определения } 0 \leq \theta \leq 2\pi, \text{ область значений } [-1, 1]).$$

$$\sin \theta = \frac{\text{Противолежащая сторона}}{\text{Гипотенуза}} = \frac{y}{r} \quad (\text{область определения } 0 \leq \theta \leq 2\pi, \text{ область значений } [-1, 1]).$$

$$\operatorname{tg} \theta = \frac{\sin \theta}{\cos \theta} = \frac{\text{Противолежащая сторона}}{\text{Прилежащая сторона}} = \frac{y}{x} \quad (\text{область определения } -\pi/2 < \theta < \pi/2, \text{ область значений } (-\infty, \infty)).$$

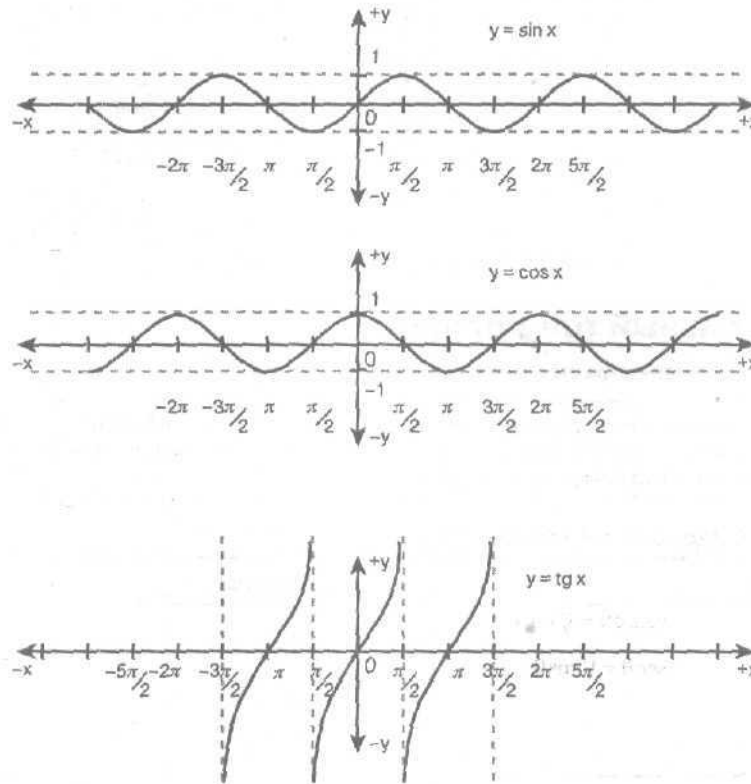


Рис. 4.14. Графики основных тригонометрических функций

На рис. 4.14 показаны графики этих функций. Обратите внимание на периодичность функций и на то, что период синуса и косинуса равен  $2\pi$ , а тангенса —  $\pi$ . Кроме того, значение тангенса угла, стремящегося к  $\pm\pi/2$ , стремится к  $\pm\infty$ .



Термины “область определения” и “область значений” по сути описывают **допустимые** входные и выходные данные функций.

## Обратные тригонометрические функции

Нередко приходится сталкиваться с задачей поиска угла, если известно значение тригонометрической функции, примененное к этому углу. Для этого используются *обратные тригонометрические функции*, которые носят те же имена, но с приставкой *арс*, либо *обозначаются* теми же именами, что и исходные функции, но с верхним индексом  $-1$ .

$$\theta = \arccos x = \cos^{-1} x, \text{ где } 0 \leq \theta \leq 2\pi, -1 \leq x \leq 1;$$

$$\theta = \arcsin x = \sin^{-1} x, \text{ где } 0 < \theta \leq 2\pi, -1 \leq x \leq 1;$$

$$\theta = \operatorname{arctg} x = \operatorname{tg}^{-1} x, \text{ где } -\pi/2 < \theta < \pi/2, -\infty < x < +\infty.$$

НА ЗАМЕТКУ

Конечно, обычно компьютер не может корректно представить бесконечность, но в контексте программирования игр нас вполне устроит число, заведомо во много раз большее, чем любое реально используемое при расчетах. Например, если игра работает с сеткой  $1000 \times 1000$ , то значение  $100000000$  вполне можно считать бесконечным.

Использовать обратные тригонометрические функции очень легко — надо просто подставить значение  $x$  и получить  $\theta$  (вычисления пусть проводит компьютер). Но здесь имеется одна неприятность — стандартные тригонометрические функции вычисляются и так достаточно медленно, но обратные тригонометрические функции **вычисляются** на порядок медленнее! Так что перед тем как проводить вычисления, убедитесь, что вам действительно надо знать значение угла. Очень часто оказывается, что достаточно знания  $x$ . Конечно, если вам **действительно** нужен угол — его можно найти быстрее, применяя таблицы поиска и интерполяцию.

## Тригонометрические тождества

Имеется множество различных тригонометрических тождеств, для доказательства которых нужна отдельная (и немалая) книга, так что ограничимся только одной таблицей 4.5, но такой, которую программист игр должен знать **наизусть**.

Таблица 4.5. Полезные тригонометрические тождества

### Обратные функции

Косеканс:  $\operatorname{cosec} \theta = 1/\sin \theta$

Секанс:  $\sec \theta = 1/\cos \theta$

Котангенс:  $\operatorname{ctg} \theta = 1/\operatorname{tg} \theta$

### Теорема Пифагора

$$\sin^2 \theta + \cos^2 \theta = 1$$

### Формула приведения

$$\sin \theta = \cos(\theta - \pi/2)$$

### Тождества четности

$$\sin(-\theta) = -\sin \theta$$

$$\cos(-\theta) = \cos \theta$$

## Формулы сложения

$$\sin(\theta \pm \phi) = \sin \theta \cos \phi \pm \cos \theta \sin \phi$$

$$\cos(\theta \pm \phi) = \cos \theta \cos \phi \mp \sin \theta \sin \phi$$

Конечно, вы можете выводить новые тождества на основе приведенных до тех пор, пока перед глазами не появятся зеленые круги. Все эти тождества нужны для того, чтобы преобразовать сложные формулы с участием тригонометрических формул в более простые, требующие меньшего количества вычислений. Так что если в вашем алгоритме встречается масса тригонометрических функций, постарайтесь максимально упростить формулы (возможно, с математическим справочником в руках), чтобы пришлось выполнять как можно меньше вычислений. Всегда помните: главное — скорость, скорость и еще раз скорость!

## СЕКРЕТ

Когда мы перейдем к матричным преобразованиям, мы сможем объединять несколько операций в одну. Заметим, что операция поворота может представлять собой объединение нескольких поворотов в один, а так как каждый поворот выражается через синусы и косинусы, то при таком объединении имеет прямой смысл упростить результирующую матрицу при помощи тригонометрических тождеств вместо применения перемножения матриц для общего случая.

## Векторы

Векторы представляют собой фундамент, на котором строятся все трехмерные алгоритмы, и поэтому отличное знание их для программиста игр — насущная необходимость. В основе своей векторы — не более чем набор из двух или большего числа компонентов, которые при наличии двух или трех компонентов представляют направленные отрезки на плоскости или в пространстве. Эти отрезки определяются своими начальными и конечными точками, как показано на рис. 4.15. Как видно на рисунке, вектор определяется двумя точками — начальной  $P_1$  и конечной  $P_2$ . Для вычисления вектора надо просто вычесть его начальную точку из конечной:

$$\mathbf{u} = P_2 - P_1 = (x_2 - x_1, y_2 - y_1) = \langle u_x, u_y \rangle.$$

Другой способ обозначения вектора из точки  $P_1$  в точку  $P_2$  состоит в том, чтобы записать конечные точки рядом и провести над ними черту:  $\overline{P_1 P_2}$ , которая означает, что это вектор из точки  $P_1$  в точку  $P_2$ , а не просто две отдельные точки. Вектор обычно записывается при помощи полужирного шрифта, а его компоненты помещаются в угловых скобках, например:  $\langle u_x, u_y \rangle$ .

Итак, вектор представляет направленный отрезок от одной точки до другой, но такой отрезок может в свою очередь представлять множество концепций — таких как *скорость* или *ускорение*. Однако будьте осторожны: будучи определен, вектор всегда отсчитывается относительно начала координат. Это означает, что после того, как вы создадите вектор из точки  $P_1$  в точку  $P_2$ , начальная его точка в пространстве векторов всегда будет находиться в точке  $(0, 0)$  или  $(0, 0, 0)$  в трехмерном случае. Это не так важно, поскольку обо всем позаботится математика, но когда вы стараетесь понять, что такое вектор, об этом следует помнить. Вектор состоит из двух чисел (трех в трехмерном случае), так что фактически он определяет одну (конечную) точку в двумерном или трехмерном пространстве. Следовательно, начальная точка должна быть определена заранее, и это — начало координат. Это

не так важно, поскольку вы не можете выполнять различные геометрические операции с вектором непосредственно, это всего лишь означает, что вы не должны забывать, что же такое вектор на самом деле.

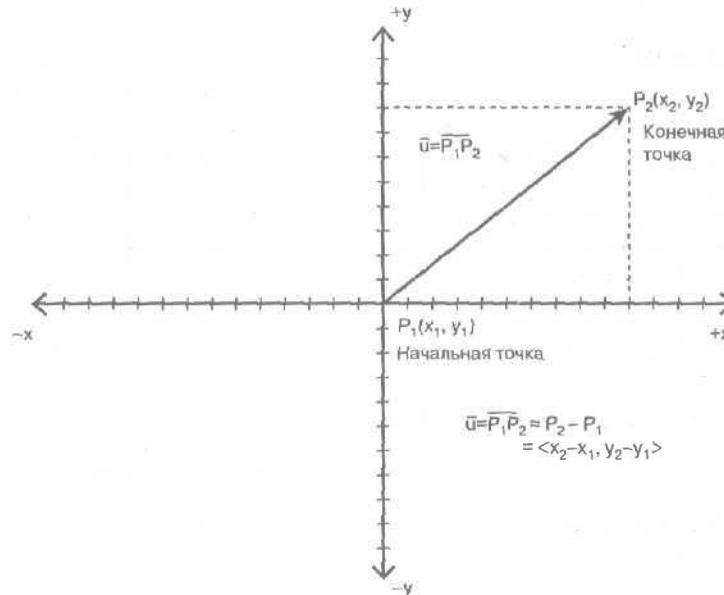


Рис. 4.15. Определение вектора

Самое ценное в векторах — это операции, которые мы можем выполнять над ними. Поскольку вектор всего лишь упорядоченное множество чисел, к нему можно применить массу стандартных математических операций, просто применяя их к каждому компоненту вектора независимо.

#### НА ЗАМЕТКУ

Векторы могут иметь любое количество компонентов. Обычно в компьютерной графике мы работаем с двумерными и трехмерными векторами, т.е. векторами вида  $a = \langle x, y \rangle$  или  $b = \langle x, y, z \rangle$ . N-мерный вектор имеет вид  $c = \langle c_1, c_2, \dots, c_n \rangle$ . Такие векторы используются для представления множеств переменных, а не реального геометрического пространства, поскольку после трехмерного пространства мы попадаем в воображаемое гиперпространство, в котором измерений больше трех.

## Длина вектора

Первое, что нас интересует при работе с векторами, это — как вычислить длину вектора, т.е. его величину. Длина вектора называется его *нормой*, и обозначается с помощью двух вертикальных линий по сторонам от вектора, так что  $|u|$  следует читать как "длина  $u$ ". Длина вычисляется как расстояние от начала координат до вершины вектора, т.е. в общем случае, согласно теореме Пифагора — как квадратный корень из суммы квадратов его компонентов.

**Уравнение 4.7.** Длина двумерного вектора  $u = \langle u_x, u_y \rangle$

$$|u| = \sqrt{u_x^2 + u_y^2}.$$

#### Уравнение 4.8. Длина трехмерного вектора $\mathbf{u} = \langle u_x, u_y, u_z \rangle$

$$|\mathbf{u}| = \sqrt{u_x^2 + u_y^2 + u_z^2}.$$

### Нормализация

После определения длины вектора с ним можно выполнить интересную операцию — *нормализовать* его, или, другими **словами**, получить из него вектор с тем же направлением, но длиной 1.0. Такой единичный вектор обладает массой полезных свойств, как и скалярная величина 1.0. Например, бывают **ситуации**, когда мы выполняем над вектором массу **операций**, но нас интересует только направление получившегося вектора, а не его длина. В таком случае нам нужен единичный вектор, который можно получить путем **нормализации** вектора, полученного в результате вычислений. Если нам дан вектор  $\mathbf{p}$ , то соответствующий нормализованный вектор обозначается как  $\hat{\mathbf{p}}$  (впрочем, в данной книге зачастую будет использоваться другое, более простое обозначение  $\mathbf{p}'$ ). Вычисляется нормализованный вектор следующим образом.

#### Уравнение 4.9. Нормализация вектора

$$\mathbf{p}' = \mathbf{p} / |\mathbf{p}|.$$

Как видите, это очень просто. Нормализованный вектор — это исходный вектор, деленный на свою длину.

### Умножение на скаляр

Первая операция, которую вы можете захотеть выполнить с вектором, — это его масштабирование. Представим, например, что у нас есть вектор, представляющий скорость, и мы хотим увеличить (или уменьшить) ее. Масштабирование вектора выполняется путем умножения каждого компонента вектора на одно скалярное число. Пусть, например,  $\mathbf{u} = \langle u_x, u_y \rangle$  и  $k$  — действительная константа. Тогда  $k \cdot \mathbf{u} = k \cdot \langle u_x, u_y \rangle = \langle k \cdot u_x, k \cdot u_y \rangle$ . На рис. 4.16 операция масштабирования **показана** графически. Кроме того, вы можете изменить направление вектора на **противоположное** умножением на  $-1$ , как показано на рис. 4.17.

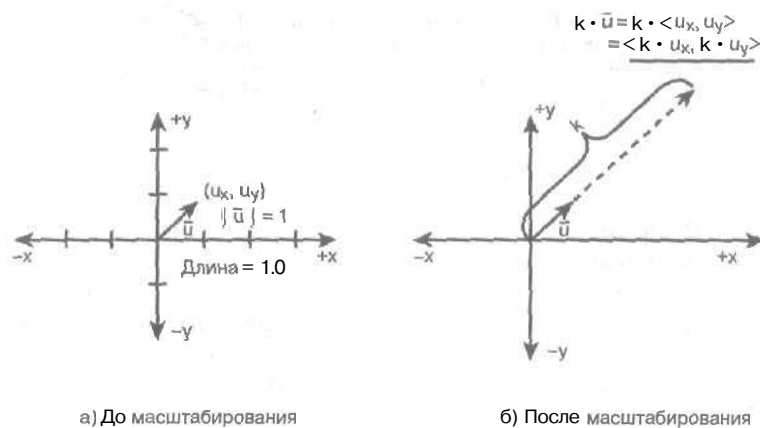


Рис. 4.16. Масштабирование вектора

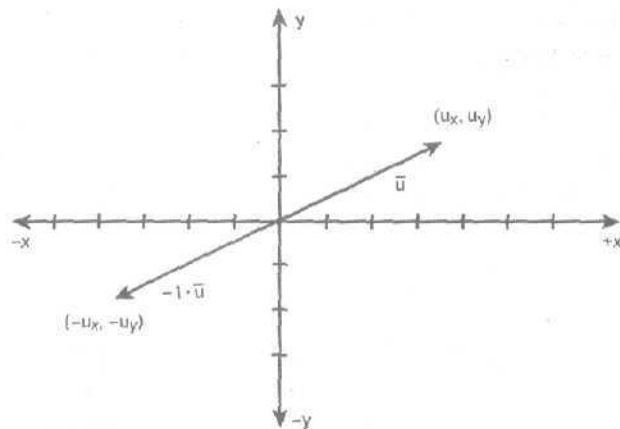


Рис. 4.17. Инверсия вектора

## Сложение векторов

Для сложения двух (или большего числа) векторов следует сложить их соответствующие компоненты. На рис. 4.18 графически показано сложение векторов в двумерном случае (сложение в трехмерном случае превышает мои художественные способности). Обратите внимание, как выполняется сложение векторов геометрически: вектор  $v$  переносится в конечную точку вектора  $u$ , после чего результатом сложения векторов  $v$  и  $u$  является вектор из начальной точки вектора  $u$  в конечную точку вектора  $v$ . Геометрически это эквивалентно следующей математической операции:

$$\mathbf{u} + \mathbf{v} = \langle u_x, u_y \rangle + \langle v_x, v_y \rangle = \langle u_x + v_x, u_y + v_y \rangle.$$

Для сложения нескольких векторов на бумаге их надо просто выстроить один за другим, чтобы начальная точка следующего вектора совпадала с конечной точкой предыдущего; тогда суммарный вектор представляет собой вектор от начальной точки первого из векторов-слагаемых к конечной точке последнего вектора.

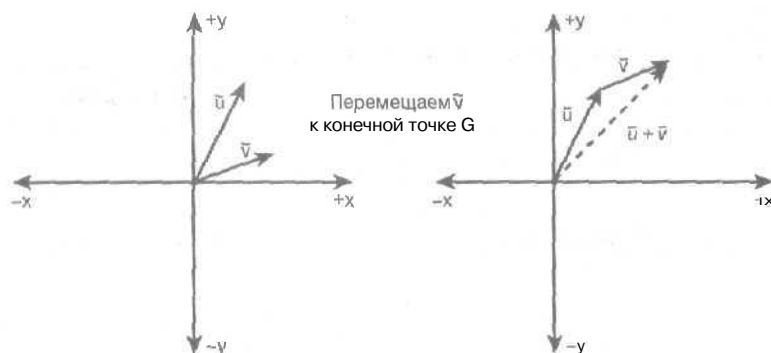


Рис. 4.18. Сложение векторов

## Вычитание векторов

Вычитание векторов на самом деле представляет собой сложение вектора с вектором, указывающим в противоположном направлении, однако гораздо нагляднее выглядит

графическое вычитание векторов (рис. 4.19). Разность  $\mathbf{u} - \mathbf{v}$  находится путем черчения вектора от конечной точки  $\mathbf{v}$  к конечной точке  $\mathbf{u}$ , что математически эквивалентно следующему:

$$\mathbf{u} - \mathbf{v} = \langle u_x, u_y \rangle - \langle v_x, v_y \rangle = \langle u_x - v_x, u_y - v_y \rangle,$$

что, конечно же, запомнить гораздо проще. Тем не менее, просто необходимо знать, как выполняется сложение и вычитание векторов *вручную*, на бумаге, поскольку это очень пригодится вам при работе с графическими алгоритмами — поверьте на слово!

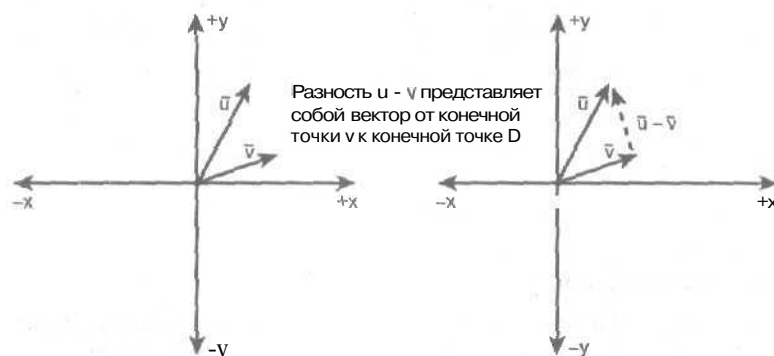


Рис. 4.19. Вычитание векторов

## Скалярное произведение

Вы можете спросить — а нельзя ли перемножить два вектора? Конечно, можно, еще и не одним способом. Но простейший способ (обозначаемый символом  $\otimes$ ), состоящий в попарном произведении компонентов ( $\mathbf{u} \otimes \mathbf{v} = \langle u_x \cdot v_x, u_y \cdot v_y \rangle$ ), особой пользы не приносит. Гораздо больше толку от произведения, определяемого следующим уравнением.

Уравнение 4.10. Скалярное произведение

$$\mathbf{u} \cdot \mathbf{v} = u_x \cdot v_x + u_y \cdot v_y.$$

Такое произведение, которое вычисляется как сумма попарных произведений компонентов векторов, называется *скалярным произведением* (поскольку дает в результате не вектор, а скаляр). Вы можете спросить — какой в нем толк? Ведь в результате не получается даже вектор! Оказывается, что скалярное произведение определяется еще и следующим образом.

Уравнение 4.11. Скалярное произведение и его связь с углом между векторами

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos \theta.$$

Говоря обычным языком, это означает, что скалярное произведение *двух* векторов равно произведению их длин на косинус угла между ними. Таким образом, комбинируя уравнения 4.10 и 4.11, мы получим, что

$$u_x \cdot v_x + u_y \cdot v_y = |\mathbf{u}| |\mathbf{v}| \cos \theta.$$

Это очень интересная формула, которая позволяет нам вычислять угол между двумя векторами (рис. 4.20). Если вы еще не сообразили, как это делается, — для вас придется расписать это поподробнее.

#### Уравнение 4.12. Вычисление угла между двумя векторами

$$\theta = \arccos\left(\frac{u_x \cdot v_x + u_y \cdot v_y}{|u| \cdot |v|}\right) = \arccos\left(\frac{u_x \cdot v_x + u_y \cdot v_y}{\sqrt{u_x^2 + u_y^2} \sqrt{v_x^2 + v_y^2}}\right).$$

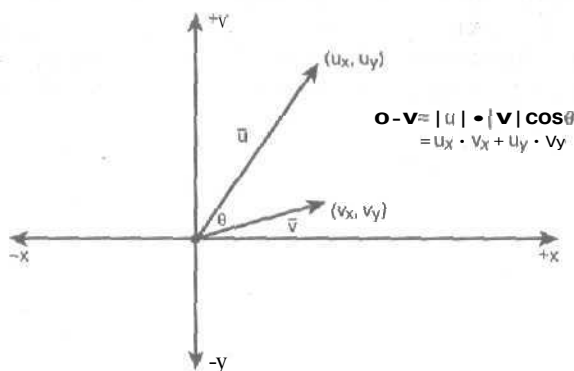


Рис. 4.20. Скалярное произведение

Заметим, что если  $\mathbf{u}$  и  $\mathbf{v}$  — единичные векторы, то угол между ними определяется совсем просто —  $\theta = \arccos(\mathbf{u} \cdot \mathbf{v})$ .

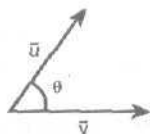
Итак, из определения скалярного произведения векторов вытекает несколько интересных фактов (проиллюстрированных на рис. 4.21).

Факт 1. Если угол между векторами  $\mathbf{u}$  и  $\mathbf{v}$  равен  $90^\circ$  (прямой), то  $\mathbf{u} \cdot \mathbf{v} = 0$ .

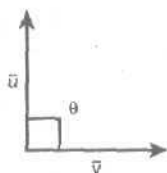
Факт 2. Если угол между векторами  $\mathbf{u}$  и  $\mathbf{v}$  меньше  $90^\circ$  (острый), то  $\mathbf{u} \cdot \mathbf{v} > 0$ .

Факт 3. Если угол между векторами  $\mathbf{u}$  и  $\mathbf{v}$  больше  $90^\circ$  (тупой), то  $\mathbf{u} \cdot \mathbf{v} < 0$ .

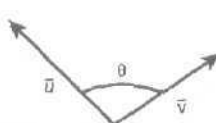
Факт 4. Если векторы  $\mathbf{u}$  и  $\mathbf{v}$  равны, то  $\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}|^2 = |\mathbf{v}|^2$ .



$\theta < 90^\circ$ , острый угол  
 $\mathbf{u} \cdot \mathbf{v} > 0$



$\theta = 90^\circ$ , прямой угол  
 $\mathbf{u} \cdot \mathbf{v} = 0$



$\theta > 90^\circ$ , тупой угол  
 $\mathbf{u} \cdot \mathbf{v} < 0$

Рис. 4.21. Углы между векторами и их скалярное произведение

Скалярное произведение может использоваться и во многих других вычислениях. Одним из примеров может служить вычисление компоненты вектора вдоль направления данного тестового вектора (т.е. вычисление **проекции** одного вектора на другой), что проиллюстрировано на рис. 4.22. Предположим, **например**, что у нас есть вектор  $v$ , представляющий траекторию одного персонажа игры, и вектор  $u$ , представляющий траекторию другого персонажа. Имеется множество ситуаций, когда требуется узнать величину компонента  $u$  в направлении  $v$ , другими словами, проекцию вектора  $u$  на вектор  $v$ , которая обозначается как  $\text{Proj}_v u$ . Эту проекцию можно вычислить с использованием **скалярного** произведения.

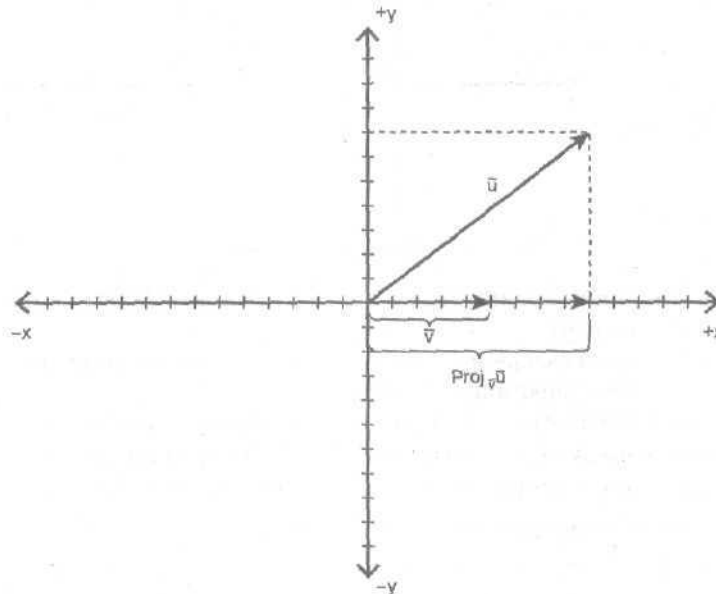


Рис. 4.22. Проекция вектора  $u$  на вектор  $v$

На рис. 4.22 показаны векторы  $u$  и  $v$ , и мы хотим определить длину проекции вектора  $u$  на вектор  $v$  (нас может интересовать и длина перпендикуляра к вектору  $v$ ), а затем, если нам нужна векторная величина, мы можем умножить полученное значение на единичный вектор  $\hat{v}$  в направлении  $v$ . При вычислениях можно использовать скалярное произведение  $u \cdot v$ , но длина вектора  $v$  при этом должна быть равна 1.0 — т.е. вектор  $v$  должен быть нормализован. Таким образом, окончательное уравнение для вектора проекции  $u$  на вектор  $v$  выглядит следующим образом.

#### Уравнение 4.13. Проекция вектора $u$ на вектор $v$

$$\text{Proj}_v u = \frac{(u \cdot v)}{\|v\|^2} \cdot v$$

#### Свойства скалярного произведения

Обычное умножение скаляров подчиняется ассоциативному, дистрибутивному и коммутативному законам, однако в векторном пространстве эти законы оказываются справедливыми не всегда. Что касается конкретно скалярного произведения векторов, то

оно подчиняется следующим правилам: для векторов  $u$ ,  $v$  и  $w$  и скаляра  $k$  справедливы соотношения

1.  $u \cdot v = v \cdot u$
2.  $u \cdot (v + w) = (u \cdot v) + (u \cdot w)$
3.  $k \cdot (u \cdot v) = (k \cdot u) \cdot v = (u \cdot (k \cdot v))$

Заметим также, что скалярное умножение имеет более высокий приоритет, чем сложение.

## Векторное умножение

Следующий тип умножения, которое может быть выполнено с векторами — *векторное умножение*, которое имеет смысл только для векторов с тремя или большим количеством компонент (мы будем рассматривать векторы в трехмерном пространстве). Пусть даны векторы  $u = \langle u_x, u_y, u_z \rangle$  и  $v = \langle v_x, v_y, v_z \rangle$ . Их векторное произведение  $u \times v$  определяется следующим образом.

### Уравнение 4.14. Векторное произведение, выраженное через угол между векторами и нормальный вектор

$$u \times v = |u| \cdot |v| \cdot \sin \theta \cdot n.$$

Давайте внимательно рассмотрим эту формулу.  $|u|$  означает длину вектора  $u$ ,  $|v|$  — длину вектора  $v$ ,  $\sin \theta$  — синус угла между этими векторами. Все это скалярные величины, так что и произведение  $|u| \cdot |v| \cdot \sin \theta$  также является скаляром. Далее это число умножается на  $n$ . Что же такое  $n$ ? Это единичный нормальный вектор, т.е. вектор, перпендикулярный векторам  $u$  и  $v$  и имеющий длину 1.0. На рис. 4.23 векторное произведение изображено графически.

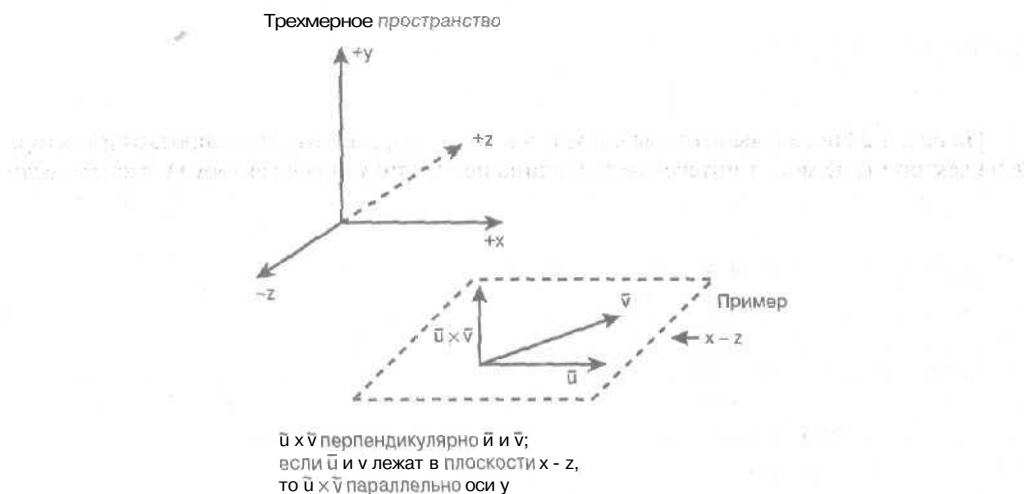


Рис. 4.23. Векторное произведение

Итак, векторное произведение содержит как информацию об угле между векторами-сомножителями, так и о нормальном к ним векторе. Но как же вычислить этот нормаль-

ный вектор? Ответ можно получить при помощи другого определения векторного произведения.

Для этого другого определения векторного произведения потребуются матрицы, так что если вы с ними незнакомы, просто доверьтесь мне. Итак, чтобы вычислить векторное произведение  $u \times v$ , надо построить матрицу

$$\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{bmatrix},$$

где  $\mathbf{i}, \mathbf{j}, \mathbf{k}$  — единичные векторы, параллельные осям  $x, y$  и  $z$ , соответственно.

После этого векторное произведение вычисляется как определитель этой матрицы.

#### Уравнение 4.15. Вычисление векторного произведения $u \times v$

$$\begin{aligned} u \times v &= (u_y \cdot v_z - u_z \cdot v_y) \cdot \mathbf{i} + (u_z \cdot v_x - u_x \cdot v_z) \cdot \mathbf{j} + (u_x \cdot v_y - u_y \cdot v_x) \cdot \mathbf{k} = \\ &= (u_y \cdot v_z - u_z \cdot v_y, u_z \cdot v_x - u_x \cdot v_z, u_x \cdot v_y - u_y \cdot v_x). \end{aligned}$$

На практике векторное произведение используется в первую очередь для того, чтобы получить вектор, нормальный обоим векторам-сомножителям.

#### Свойства векторного умножения

Для векторов  $u, v$  и  $w$  и скаляра  $k$  справедливы следующие соотношения:

1.  $u \times v = -(v \times u)$  (очень важно!)
2.  $u \times (v + w) = (u \times v) + (u \times w)$
3.  $(u + v) \times w = (u \times w) + (v \times w)$
4.  $k \cdot (u \times v) = (k \cdot u) \times v = u \times (k \cdot v)$

#### Нулевой вектор

Хотя вряд ли вам придется часто сталкиваться с нулевым вектором, тем не менее, он существует. Это вектор, который имеет нулевую длину и не имеет направления, и технически представляет собой просто точку. В двумерном случае это вектор  $\langle 0, 0 \rangle$ , в трехмерном —  $\langle 0, 0, 0 \rangle$  и т.д.

#### Радиус-вектор и вектор перемещения

Следующая тема, которую я бы хотел затронуть, — это радиус-векторы и векторы перемещения. Эти концепции оказываются весьма полезны при вычерчивании геометрических объектов наподобие линий, кривых и т.п. Взгляните на рис. 4.24 — на нем изображен радиус-вектор, который может использоваться для представления отрезка. Отрезок проведен из точки  $p_1$  в точку  $p_2$ ; вектор  $v_d$  представляет собой вектор перемещения из  $p_1$  в  $p_2$ , а  $\hat{v}$  — единичный вектор в направлении от  $p_1$  к  $p_2$ . Мы строим вектор  $r$  для черчения отрезка, который математически выглядит следующим образом:  $r = p_1 + t \cdot \hat{v}$ , где  $t$  — параметр, изменяющийся от 0 до  $|v_d|$ . При  $t = 0$   $r = p_1 + 0 \cdot \hat{v} = p_1$ , т.е. вектор  $r$  указывает на начало отрезка, а при  $t = |v_d|$

$$\begin{aligned}\mathbf{p} &= \mathbf{p}_1 + |\mathbf{v}_d| \cdot \hat{\mathbf{v}} = \mathbf{p}_1 + \mathbf{v}_d = \\ &= \langle p_{1x} + v_{dx}, p_{1y} + v_{dy} \rangle = \mathbf{p}_2.\end{aligned}$$

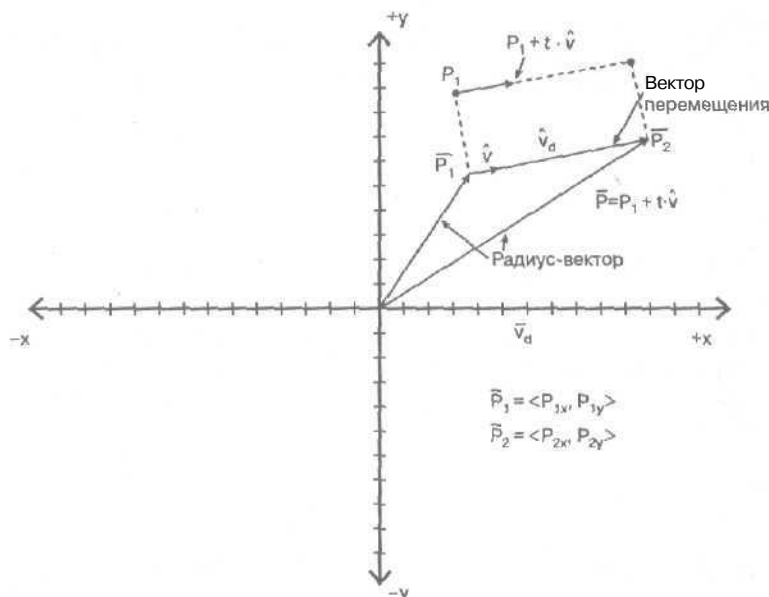


Рис. 4.24. Радиус-векторы вектор перемещения

## Векторы как линейные комбинации

Как было показано при рассмотрении векторного произведения, векторы могут быть записаны при помощи следующего обозначения

$$\mathbf{u} = u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k},$$

где  $\mathbf{i}$ ,  $\mathbf{j}$  и  $\mathbf{k}$  — единичные векторы, параллельные осям  $x$ ,  $y$  и  $z$ . В этом нет никакого колдовства — это просто еще один способ записи векторов, который вы должны знать. Все операции при этом остаются такими же, как и раньше. Например, пусть  $\mathbf{u} = 3\mathbf{i} + 2\mathbf{j} + 3\mathbf{k}$  и  $\mathbf{v} = -3\mathbf{i} - 5\mathbf{j} + 12\mathbf{k}$ ; тогда

$$\begin{aligned}\mathbf{u} + \mathbf{v} &= 3\mathbf{i} + 2\mathbf{j} + 3\mathbf{k} - 3\mathbf{i} - 5\mathbf{j} + 12\mathbf{k} = \\ &= 0\mathbf{i} - 3\mathbf{j} + 15\mathbf{k} = \langle 0, -3, 15 \rangle.\end{aligned}$$

## Матрицы и линейная алгебра

Трехмерная графика представляет собой по сути не что иное, как множество математических операций над большим количеством списков чисел — так что использование матриц упрощает процесс преобразования и представления таких типов данных.

*Матрица* представляет собой прямоугольный массив с заданным числом строк и столбцов. Обычно, когда говорится, что матрица имеет размер  $m \times n$ , подразумевается, что у нее  $m$  строк и  $n$  столбцов. Размер  $m \times n$  матрицы называется также ее *размерностью*. Например, приведенная далее матрица  $A$  имеет размерность  $2 \times 2$ :

$$A = \begin{bmatrix} 1 & 4 \\ 9 & -1 \end{bmatrix}.$$

Обратите внимание, что я использовал полужирную прописную букву для обозначения матрицы. Обычно в литературе (и в данной книге) используется именно такое обозначение, но мне встречалось обозначение матрицы, например, с использованием курсива. В приведенном примере первая строка матрицы —  $\langle 1 \ 4 \rangle$ , а вторая —  $\langle 9 \ -1 \rangle$ . Вот пример матриц  $3 \times 2$  и  $2 \times 3$ :

$$B = \begin{bmatrix} 5 & 6 \\ 2 & 3 \\ 100 & -7 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 5 & 0 \\ -8 & 12 & 4 \end{bmatrix}.$$

Для того чтобы найти  $\langle i, j \rangle$ -й элемент матрицы, вы просто смотрите на значение на пересечении  $i$ -й строки и  $j$ -го столбца. Однако здесь есть одна тонкость — дело в том, что в математике отсчет номеров строк и столбцов начинается с 1, а не с 0, как в программировании — и об этом надо не забывать. Мы же будем начинать отсчет с нуля, что позволяет сделать работу с матрицами в C/C++ более естественной. Вот, например, как выглядят элементы матрицы  $3 \times 3$ :

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}.$$

Достаточно просто, не правда ли? Теперь, когда вы познакомились с матрицами и используемыми при работе с ними соглашениями, вы можете спросить: и зачем они нам нужны? Матрицы — это математический инструмент для представления систем уравнений. Обычно, когда мы имеем дело с системой уравнений, например:

$$3x + 2y = 1$$

$$4x - 9y = 9$$

нам приходится неоднократно указывать все переменные. Но вы и так знаете, что мы работаем с переменными  $(x, y)$ . Так зачем все время напоминать об этом? Почему бы не создать компактную форму записи? Такая форма есть — с использованием матриц. В рассмотренной системе уравнений есть три различных множества значений, перечисленные ниже, которые мы можем разместить в матрицах, и работать со всеми ими вместе или по отдельности.

1. Коэффициенты при переменных:

$$3x + 2y = 1,$$

$$4x - 9y = 9,$$

$$A = \begin{bmatrix} 3 & 2 \\ 4 & -9 \end{bmatrix}.$$

Размерность матрицы  $A$  —  $2 \times 2$ .

2. Переменные;

$$3x + 2y = 1,$$

$$4x - 9y = 9,$$

$$X = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Размерность матрицы  $X$  —  $2 \times 1$ .

3. Константы в правой части системы уравнений:

$$3x + 2y = 1,$$

$$4x - 9y = 9,$$

$$B = \begin{bmatrix} 1 \\ 9 \end{bmatrix}.$$

Размерность матрицы  $B$  —  $2 \times 1$ .

После того как введены все эти матрицы, мы можем, например, работать с матрицей коэффициентов  $A$  отдельно от остальных данных. Кроме того, мы можем записать систему уравнений в виде

$$A \cdot X = B,$$

что после применения правил умножения матриц дает нам исходную систему уравнений

$$3x + 2y = 1,$$

$$4x - 9y = 9.$$

Но как же выполняется умножение матриц? Мы уже знакомы с умножением векторов, теперь пора узнать, как эта операция выполняется с **матрицами**, однако сначала давайте познакомимся с концепцией единицы в контексте матриц.

## Единичная матрица

Первое, что нам надо сделать в любой математической системе, — это определить 1 и 0. В матричной математике имеются аналоги обоих этих значений. Аналог 1 называется *единичной матрицей*  $I$ , которая образуется путем размещения единиц на главной **диагонали** матрицы, и нулей в остальных местах. Единичная матрица обладает тем **свойством**, что  $A \cdot I = I \cdot A = A$  (матрицы  $I$  и  $A$  должны иметь одинаковый размер).

Кроме того, поскольку матрицы могут быть самых разных размеров, очевидно, что имеется бесконечное количество единичных матриц, но на них накладывается одно ограничение — все единичные матрицы должны быть *квадратными*, т.е. размера  $m \times m$ , где  $m \geq 1$ . Вот несколько примеров единичных матриц:

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ размерность } 2 \times 2.$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ размерность } 3 \times 3.$$

Технически единичная матрица не является точным аналогом 1, но является таковым в смысле умножения матриц, с которым мы вскоре познакомимся.

Второй тип фундаментальной матрицы — *нулевая матрица*  $Z$ , которая **представляет** собой аналог 0 как в плане **сложения**, так и умножения. Это ни что иное, как матрица размером  $m \times n$ , заполненная нулями. Никакие другие ограничения на нулевую матрицу не накладываются. Вот примеры нулевых матриц:

$$Z_{3 \times 3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad Z_{1 \times 2} = \begin{bmatrix} 0 & 0 \end{bmatrix}.$$

Единственное интересное свойство нулевой матрицы заключается в ее тождественности 0 как в смысле сложения, так и в смысле умножения. По сути, у нулевых матриц нет никакого полезного применения.

## Сложение матриц

Сложение и вычитание матриц выполняется путем сложения или вычитания элементов двух матриц для получения результирующего элемента, находящегося в той же позиции, что и указанные элементы матриц. Единственное накладываемое **при** этом ограничение состоит в том, что матрицы, над которыми выполняются данные операции, должны иметь одинаковый размер. Вот небольшой пример.

Пусть  $A = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix}$  и  $B = \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix}$ . Тогда

$$A + B = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} + \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix} = \begin{bmatrix} 1+13 & 5+7 \\ -2+5 & 0-10 \end{bmatrix} = \begin{bmatrix} 14 & 12 \\ 3 & -10 \end{bmatrix} \text{ и}$$

$$A - B = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} - \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix} = \begin{bmatrix} 1-13 & 5-7 \\ -2-5 & 0+10 \end{bmatrix} = \begin{bmatrix} -12 & -2 \\ -7 & 10 \end{bmatrix}.$$

## Транспонирование матрицы

Иногда в процессе расчетов требуется *транспонированная* матрица, строки которой заменены столбцами. Например, для матрицы размером  $1 \times 3$   $M = [x \ y \ z]$  транспонированной является матрица размером  $3 \times 1$ :

$$M' = \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

В общем случае матрицы  $m \times n$  транспонированная матрица вычисляется простой заменой столбцов строками, т.е. элемент  $a_{ij}$  меняется местами с элементом  $a_{ji}$ . Ниже приведен пример транспонирования матрицы размером  $3 \times 2$ :

$$A = \begin{bmatrix} 5 & 6 \\ 2 & 3 \\ 100 & -7 \end{bmatrix}, \quad A' = \begin{bmatrix} 5 & 2 & 100 \\ 6 & 3 & -7 \end{bmatrix}.$$

Вот и все, что касается транспонирования. Транспонирование применяется, например, при обращении матриц.

## Умножение матриц

Есть два вида умножения матриц — *скалярное умножение* матриц и *матричное умножение*. Скалярное умножение обозначает просто умножение матрицы на скаляр и выполняется путем умножения каждого элемента матрицы на этот скаляр (матрица при этом может иметь любой размер), т.е. после умножения на скаляр  $k$  каждый элемент матрицы  $a_{ij}$  становится равным  $k \cdot a_{ij}$ . Вот как выглядит скалярное умножение матрицы  $3 \times 3$ :

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, \quad k \cdot A = \begin{bmatrix} k \cdot a_{00} & k \cdot a_{01} & k \cdot a_{02} \\ k \cdot a_{10} & k \cdot a_{11} & k \cdot a_{12} \\ k \cdot a_{20} & k \cdot a_{21} & k \cdot a_{22} \end{bmatrix}.$$

А вот — конкретный пример скалярного умножения.

$$3 \cdot \begin{bmatrix} 1 & 4 \\ -2 & 6 \end{bmatrix} = \begin{bmatrix} 3 \cdot 1 & 3 \cdot 4 \\ 3 \cdot (-2) & 3 \cdot 6 \end{bmatrix} = \begin{bmatrix} 3 & 12 \\ -6 & 18 \end{bmatrix}.$$

$\sum_{\alpha}^{\infty}$

Скалярное является корректной операцией для матричных уравнений, если вы умножаете на один и тот же скаляр обе части уравнения (поскольку умножение всех коэффициентов любой системы линейных уравнений на одно и то же число не влияет на ее решение).

Второй вид умножения — матричное. Математически матричное умножение несколько сложнее, но его можно более-менее корректно рассматривать как применение матрицы-оператора к другой матрице.

Две матрицы могут быть перемножены тогда и только тогда, когда они имеют одинаковый внутренний размер, т.е. если матрица  $A$  имеет размер  $m \times n$ , а матрица  $B$  — размер  $p \times q$ , то умножение  $A$  на  $B$  возможно только тогда, когда  $n = p$ . Величины  $m$  и  $q$  могут быть при этом любыми. Например, возможно умножение матриц размером  $2 \times 2$  на  $2 \times 2$ ,  $3 \times 2$  на  $2 \times 3$  или  $4 \times 4$  на  $4 \times 5$ , но не умножение  $3 \times 3$  на  $2 \times 4$ . Умножение матриц  $m \times n$  на  $n \times p$  дает матрицу  $m \times p$  — например, умножение матриц размером  $3 \times 4$  на  $4 \times 5$  дает матрицу размера  $3 \times 5$ .

Произведение матриц легко выразить формулой  $c_{ij} = \sum_k a_{ik} b_{kj}$ , но очень сложно описать словами. Чтобы понять приведенную формулу, рассмотрите рис. 4.25.

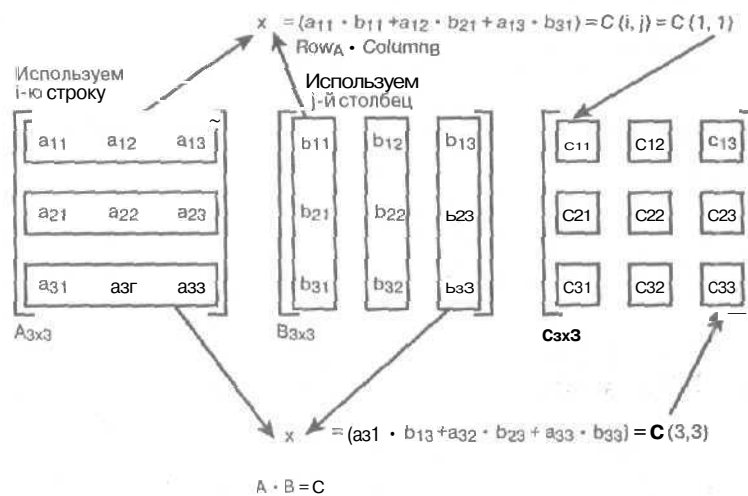


Рис. 4.25. Механизм матричного умножения

Для заданных матриц  $A$  и  $B$ , для поиска элементов их матрицы-произведения надо взять строку матрицы  $A$  и столбец матрицы  $B$  и просуммировать произведения их элементов (получить скалярное произведение векторов, представляющих строку матрицы  $A$  и столбец матрицы  $B$ ). Вот пример перемножения реальных матриц:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 5 \\ 6 & 4 \end{bmatrix},$$

$$C = A \cdot B = \begin{bmatrix} 1 \cdot 3 + 2 \cdot 6 & 1 \cdot 5 + 2 \cdot 4 \\ 3 \cdot 3 + 4 \cdot 6 & 3 \cdot 5 + 4 \cdot 4 \end{bmatrix} = \begin{bmatrix} 15 & 13 \\ 27 & 31 \end{bmatrix}.$$

Теперь вы знаете, как выполняется матричное умножение. Осталось разобраться с тем, каким правилам оно подчиняется.

## Свойства матричных операций

Так же как и скалярное и векторное произведения векторов, матричные операции обладают некоторыми свойствами, перечисленным ниже:

1.  $A + B = B + A$  (свойство коммутативности сложения).
2.  $A + (B + C) = (A + B) + C$  (свойство ассоциативности сложения).
3.  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$  (свойство ассоциативности умножения).
4.  $A \cdot (B + C) = A \cdot B + A \cdot C$  (свойство дистрибутивности).
5.  $k \cdot (A + B) = k \cdot A + k \cdot B$  (свойство дистрибутивности).
6.  $(A + B) \cdot C = A \cdot C + B \cdot C$  (свойство дистрибутивности).
7.  $A \cdot I = I \cdot A = A$  (умножение на единичную матрицу).

Во всех формулах сложение “+” можно заменить вычитанием “−”. Обратите внимание, что в общем случае  $A \cdot B \neq B \cdot A$ , т.е. для матриц свойство коммутативности не выполняется и порядок сомножителей при умножении очень важен,

НА ЗАМЕТКУ

$A \cdot B = B \cdot A$ , если один из сомножителей — единичная матрица.

## Обращение матриц и решение систем линейных уравнений

Теперь, когда вы знаете, как создавать матрицы и работать с ними, вам предстоит познакомиться с еще одним математическим объектом — *мультипликативной обратной матрицей*. Что это такое? Если у нас есть скаляр  $x$  и мы хотим найти такое число  $x^{-1}$ , что  $x \cdot x^{-1} = 1$ , то это число  $x^{-1}$  и есть мультипликативное обратное (или просто обратное) к числу  $x$ . В случае скаляров все просто. Если  $x$  — действительное число, то  $x^{-1}$  можно вычислить как  $x^{-1} = 1/x$  для всех  $x \neq 0$ . Однако в случае матриц все не так-то просто.

Например, если у нас есть матричное уравнение  $A \cdot X = B$  и мы хотим найти  $X$ , то при наличии обратной матрицы мы можем умножить на нее обе стороны уравнения

$$(A^{-1} \cdot A) \cdot X = A^{-1} \cdot B$$

и, поскольку мы знаем, что умножение матрицы на обратную дает единичную матрицу, мы легко получаем решение

$$(I \cdot X) = X = A^{-1} \cdot B.$$

Возможность решения системы линейных уравнений совершенно необходима во многих областях программирования трехмерных игр — в графике, физическом моделировании, анимации и т.д. К сожалению, у меня нет возможности рассказать здесь о том, как искать обратную матрицу в общем случае, но я все же немного затрону тему решения систем линейных уравнений хотя бы для случая  $2 \times 2$ .

Итак, пусть дана матрица

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

Обратная к ней матрица существует, причем она является единственной, если *детерминант* (или *определитель*) матрицы не равен нулю. Определитель матрицы A (или любой матрицы 2x2) можно вычислить *следующим* образом.

#### Уравнение 4.16. Детерминант матрицы 2x2

$$\det(A) = (a \cdot d - b \cdot c).$$

Если это значение не равно нулю, то существует обратная к A матрица, которую можно вычислить *следующим* образом.

#### Уравнение 4.17. Обращение матрицы 2x2

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} d/(a \cdot d - b \cdot c) & -b/(a \cdot d - b \cdot c) \\ -c/(a \cdot d - b \cdot c) & a/(a \cdot d - b \cdot c) \end{bmatrix}.$$

Протестируем данные формулы на конкретном примере системы линейных уравнений

$$3 \cdot x + 5 \cdot y = 6,$$

$$-2 \cdot x + 2 \cdot y = 4.$$

Мы хотим найти значения x и y с использованием матриц, так что начнем с того, что определим матрицы A, B и X:

$$A = \begin{bmatrix} 3 & 5 \\ -2 & 2 \end{bmatrix}, B = \begin{bmatrix} 6 \\ 4 \end{bmatrix} \text{ и } X = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Для поиска X нам надо найти матрицу, обратную матрице A, определитель которой равен  $\det(A) = (3 \cdot 2 - (-2) \cdot 5) = 16$ . В соответствии с уравнением 4.17 имеем:

$$A^{-1} = \begin{bmatrix} 2/16 & -5/16 \\ 2/16 & 3/16 \end{bmatrix}.$$

Подставляя полученную матрицу в формулу  $X = A^{-1} \cdot B$ , получаем:

$$X = \begin{bmatrix} 2/16 & -5/16 \\ 2/16 & 3/16 \end{bmatrix} \begin{bmatrix} 6 \\ 4 \end{bmatrix} = \begin{bmatrix} (12 - 20)/16 \\ (12 + 12)/16 \end{bmatrix} = \begin{bmatrix} -8/16 \\ 24/16 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1.5 \end{bmatrix}.$$

Таким образом,  $x = -0.5$  и  $y = 1.5$ . Подставим эти значения в первое и второе уравнения исходной системы для проверки:

$$3 \cdot x + 5 \cdot y = 3 \cdot (-0.5) + 5 \cdot 1.5 = -1.5 + 7.5 = 6;$$

$$-2 \cdot x + 2 \cdot y = -2 \cdot (-0.5) + 2 \cdot 1.5 = 1 + 3 = 4.$$

Как видите, полученное решение оказалось правильным. Или вы рассчитывали на что-то иное?

## Правило Крамера

Решение больших систем уравнений с использованием правила Крамера — не совсем здравая идея ввиду огромного количества расчетов для построения *коэффициентов* обратной матрицы. Обычно для систем более чем с тремя неизвестными используются итеративные технологии *наподобие метода исключения Гаусса*.

Метод исключения Гаусса представляет собой процесс преобразования матрицы коэффициентов к треугольному виду, обеспечивающему быстрое и простое решение системы уравнений. Этот процесс основан на том факте, что любая строка матрицы может быть умножена на некоторое число, или любые строки могут быть прибавлены к данной, и это не повлияет на конечное решение системы уравнений.

Однако для системы линейных уравнений с тремя неизвестными все еще вполне применимо *правило Крамера*, о котором вы, вероятно, слышали и раньше. Правило Крамера для системы из трех уравнений выглядит следующим образом. Пусть нам дана система уравнений

$$a_{00} \cdot x + a_{01} \cdot y + a_{02} \cdot z = b_0,$$

$$a_{10} \cdot x + a_{11} \cdot y + a_{12} \cdot z = b_1,$$

$$a_{20} \cdot x + a_{21} \cdot y + a_{22} \cdot z = b_2.$$

Преобразуем ее к матричному виду:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \text{ и } X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

Теперь мы должны вычислить определитель матрицы  $A$ , что можно сделать так, как показано в уравнении 4.18а.

#### Уравнение 4.18а. Определитель матрицы 3х3

$$\det(A) = a_{00} \cdot a_{11} \cdot a_{22} + a_{01} \cdot a_{12} \cdot a_{20} + a_{02} \cdot a_{10} \cdot a_{21} - \\ - a_{02} \cdot a_{11} \cdot a_{20} - a_{01} \cdot a_{10} \cdot a_{22} - a_{00} \cdot a_{12} \cdot a_{21}.$$

(Примечание. Всего 12 умножений и 5 сложений).

Тот же определитель можно переписать по-другому, что позволит вычислить его несколько быстрее.

#### Уравнение 4.18б. Определитель матрицы 3х3, альтернативная запись

$$\det(A) = a_{00} \cdot (a_{11} \cdot a_{22} - a_{21} \cdot a_{12}) - a_{01} \cdot (a_{10} \cdot a_{22} - a_{20} \cdot a_{12}) + \\ + a_{02} \cdot (a_{10} \cdot a_{21} - a_{20} \cdot a_{11}).$$

(Примечание. Всего 9 умножений и 5 сложений).

Теперь, при наличии определителя матрицы  $A$ , правило Крамера позволяет записать решение линейной системы уравнений как набор дробей, по одной для каждой из переменных  $x$ ,  $y$  и  $z$ . Числители дробей представляют собой определитель матрицы коэффициентов  $A$ , в которой первый, второй и третий столбцы заменены столбцом  $B$  для вычисления значений  $x$ ,  $y$  и  $z$ , соответственно. Знаменатель у всех дробей одинаков и равен определителю матрицы  $A$ . Распишем сказанное более подробно:

$$x = \frac{\det \begin{pmatrix} b_0 & a_{01} & a_{02} \\ b_1 & a_{11} & a_{12} \\ b_2 & a_{21} & a_{22} \end{pmatrix}}{\det(A)}$$

(Примечание. Для поиска  $x$  первый столбец  $A$  заменен вектором  $B$ ).

$$y = \frac{\det \begin{pmatrix} a_{00} & b_{01} & a_{02} \\ a_{10} & b_{11} & a_{12} \\ a_{20} & b_{21} & a_{22} \end{pmatrix}}{\det(A)}$$

(Примечание. Для поиска  $y$  второй столбец  $A$  заменен вектором  $B$ ).

$$z = \frac{\det \begin{pmatrix} a_{00} & a_{01} & b_{02} \\ a_{10} & a_{11} & b_{12} \\ a_{20} & a_{21} & b_{22} \end{pmatrix}}{\det(A)}$$

(Примечание. Для поиска  $z$  третий столбец  $A$  заменен вектором  $B$ ).

Для вычисления определителя используется одно из уравнений 4.18. Основная идея, надеюсь, вам понятна, а соответствующий код будет написан нами позже.

## Преобразования с использованием матриц

Весь этот материал рассматривался нами, по сути, с одной главной целью — для применения матриц для двух- и трехмерных преобразований. Все, что мы хотим делать — это умножать точки, которые намерены преобразовать, на соответствующие матрицы преобразования для получения новых точек, т.е. математически

$$p' = p \cdot M,$$

где  $p'$  — преобразованная точка,  $p$  — исходная точка, а  $M$  — матрица преобразования.

Кроме того, мы можем выполнить любое количество преобразований путем последовательного умножения матриц, т.е. объединить любое количество матриц в одну путем их предварительного умножения:

$$M = M_1 \cdot M_2 \cdot M_3 \cdot \dots \cdot M_n.$$

Следующие преобразования оказываются эквивалентными:

$$p' = p \cdot M = p \cdot M_1 \cdot M_2 \cdot M_3 \cdot \dots \cdot M_n$$

Умножаете ли вы  $p$  на  $M$  или на произведение  $M_1 \cdot M_2 \cdot M_3 \cdot \dots \cdot M_n$  — результат получается один и тот же. Например, пусть у нас есть три преобразования  $M_1$ ,  $M_2$  и  $M_3$ , которые мы хотим применить к точке  $p$  для получения новой точки  $p'$ . Это могут быть, например, перенос, поворот и проекция. Эту серию преобразований можно выполнить обычным путем, по одному за раз:

$$p_1 = p \cdot M_1,$$

$$p_2 = p_1 \cdot M_2,$$

$$p' = p_2 \cdot M_3.$$

( $p_1$  и  $p_2$  — временные переменные).

Такая серия преобразований вполне корректна, но что если у нас тысячи, если не миллионы, точек, которые необходимо преобразовать? Все промежуточные вычисления кажутся излишними — и совершенно справедливо. Судите сами — ведь можно объединить все эти преобразования в одно, которое и применять ко всем необходимым точкам:

$$p_1 = p \cdot M_1,$$

$$p_2 = p_1 \cdot M_2,$$

$$p' = p \cdot (M_1 \cdot M_2 \cdot M_3).$$

Как видите, преобразование  $p$  в  $p'$  можно выразить как  $p' = p \cdot M$ , где  $M = M_1 \cdot M_2 \cdot M_3$ . Таким образом, достаточно один раз вычислить матрицу  $M$ , и использовать ее вместо вычисления двух лишних матричных произведений для каждой точки. Такой процесс объединения матриц играет важную роль в трехмерной графике, в особенности при преобразовании вершин объекта в конечные точки на экране.

НА ЗАМЕТКУ

В связи с некоммутативной природой матричного умножения, порядок применения преобразований играет важную роль:  $M_1 \cdot M_2 \cdot M_3 \neq M_3 \cdot M_2 \cdot M_1$ .

## Однородные координаты

Следующий вопрос, которого я хочу вкратце коснуться, — однородные (гомогенные) координаты. В большинстве случаев мы представляем наши трехмерные объекты как набор вершин, каждая из которых имеет три компоненты:  $p(x, y, z)$ , или как вектор  $\langle x, y, z \rangle$ , или как матрицу из одной строки  $[x \ y \ z]$  — все эти понятия эквивалентны, и разница только в использованном обозначении. Однако для того чтобы получить возможность использовать всю мощь матричных преобразований, мы должны представлять точки в двумерном или трехмерном пространстве с использованием однородных координат. При этом мы получаем возможность выполнять любые возможные преобразования с применением матриц. Для преобразования точек в двумерном или трехмерном пространстве в однородные координаты к каждой из них мы должны добавить еще одну фиктивную координату  $w$ , т.е. получить строки  $[x \ y \ w]$  для двумерного и  $[x \ y \ z \ w]$  для трехмерного случая. Для преобразования из однородных координат к нормальным мы должны выполнить деление на  $w$ . Так, например, для трехмерного случая однородные координаты  $[x \ y \ z \ w]$  превращаются в  $[x/w \ y/w \ z/w]$ .

Таким образом, при  $w = 1$  преобразование не изменяет значения  $x$ ,  $y$  и  $z$ . Тогда в чем же цель добавления  $w$ ? Ну, например,  $w$  позволяет нам представить концепцию бесконечности. Другими словами, когда знаменатель дроби стремится к нулю, частное устремляется к бесконечности. Кроме того, однородные координаты позволяют использовать однородные уравнения. Например, рассмотрим следующее линейное уравнение:

$$a \cdot x + b \cdot y + c = 0.$$

Это линейное уравнение первой степени, но два члена этого уравнения имеют степень 1, в то время как член-константа имеет степень 0. Что если мы хотим сделать все степени равными (однородными)? Мы можем привести полином к однородному виду, заменив  $x$  на  $x/w$  и  $y$  на  $y/w$ :

$$a \cdot x/w + b \cdot y/w + c = 0,$$

или, проводя очевидное преобразование:

$$a \cdot x + b \cdot y + c \cdot w = 0.$$

Теперь все члены имеют степень 1, так что полином становится однородным. Цель однородных координат состоит в том, чтобы сделать возможными любые преобразования, включая, например, перемещения. Я не буду вдаваться в подробности; просто знайте, что мы часто будем использовать фиктивную переменную  $w = 1$  только для того, чтобы иметь возможность представить все преобразования при помощи матриц. В действительности в своих структурах данных вы можете не выделять место для хранения  $w$ , поскольку в абсолютном большинстве случаев  $w = 1$ , а однородные координаты  $[x \ y \ z \ w]$  при этом становятся обычными координатами  $[x \ y \ z]$ . Здесь я всего лишь хотел показать, откуда взялся этот математический трюк.

Хотя мы еще познакомимся со всей гаммой преобразований в следующей главе, тем не менее я бы хотел вкратце рассказать об основных трехмерных преобразованиях, а именно — о перемещении, масштабировании и повороте — и пояснить, какую роль при этом играют однородные координаты.

## Применение матричных преобразований

Сейчас мы рассмотрим некоторые основные матричные преобразования, которые могут выполняться как с двумерными, так и с трехмерными данными. Все примеры приведены для трехмерных данных, однако двумерные версии в большинстве случаев очень легко получить, просто опустив  $z$ . Заметим, что во всех случаях  $w=1$ .

### Перенос

Для выполнения переноса в трехмерном случае мы должны преобразовать точку  $p(x, y, z)$  в новую точку  $p'(x + dx, y + dy, z + dz)$ , как показано на рис. 4.26, где для простоты показан перенос точки на плоскости  $xy$ . Это действие можно выполнить с помощью следующей матрицы  $4 \times 4$ :

$$M_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}.$$



Рис. 4.26. Перенос точки на плоскости

Вот как выполняется перенос точки  $p = [x \ y \ z \ 1]$  при помощи данной матрицы:

$$p' = p \cdot M_t = [x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix} = [x + dx \ y + dy \ z + dz \ 1].$$

Выполняя деление первых трех компонент на  $w=1$ , получим точку  $[(x + dx)/1 \ (y + dy)/1 \ (z + dz)/1] = p'(x + dx, y + dy, z + dz)$ .

$\sum_{a=1}^{\infty}$

В действительности при написании математической библиотеки мы не будем заниматься лишней работой и делить полученные результаты на 1. Мы просто будем использовать для большинства преобразований матрицы  $4 \times 3$  и считать  $w=1$ , но пока что мы предельно точны и аккуратны.

Итак, координаты перенесенной точки оказались именно теми, что мы и хотели получить,

### Обратный перенос

Интересно заметить, что исходя из геометрического смысла переноса, обратный перенос вычисляется просто изменением знаков  $dx$ ,  $dy$  и  $dz$  в матрице  $M_t$ :

$$M_t^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{bmatrix}$$

Давайте проверим полученный результат — перемножение матриц  $M_t$  и  $M_t^{-1}$  должно дать единичную матрицу  $4 \times 4$ , соответствующую тому, что точка остается на месте;

$$M_t \cdot M_t^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I_{4 \times 4}$$

### Масштабирование

Для масштабирования точки относительно начала координат мы должны умножить компоненты точки  $p(x, y, z)$  на множители масштабирования  $s_x$ ,  $s_y$  и  $s_z$  для осей  $x$ ,  $y$  и  $z$ , соответственно. На рис. 4.27 показан упрощенный случай масштабирования точек на плоскости. Кроме того, мы предполагаем, что никакого переноса во время масштабирования нет.

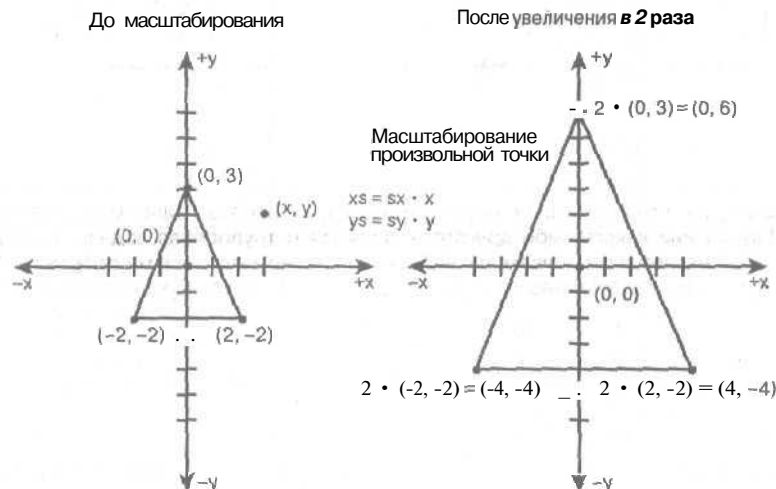


Рис. 4.27. Операция масштабирования

Вот матрица, которая позволяет выполнить операцию масштабирования:

$$M_s = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

А вот как выполняется масштабирование точки  $p = [x \ y \ z \ 1]$  при помощи данной матрицы:

$$p' = p \cdot M_s = [x \ y \ z \ 1] \cdot \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [s_x \ s_y \ s_z \ 1].$$

Выполняя деление первых трех компонент на  $w \approx 1$ , получим точку  $[(x \cdot s_x)/1 \ (y \cdot s_y)/1 \ (z \cdot s_z)/1] = p'(x \cdot s_x, y \cdot s_y, z \cdot s_z)$ . Итак, координаты перенесенной точки оказались именно теми, что мы и хотели получить.



Обратите внимание на 1 в нижнем правом углу матрицы преобразования. Технически она не нужна, поскольку мы не используем результат из четвертого столбца. Следовательно, мы только зря тратим процессорное время. С другой стороны, как уже говорилось, использование четырехкомпонентного представления для реализации однородных координат заставляет нас использовать матрицы 4x4 для математической корректности. Позже, при написании кода, мы отбросим все лишнее.

## Обратное масштабирование

Исходя из геометрического смысла масштабирования, обратное преобразование вычисляется обращением множителей  $s_x$ ,  $s_y$  и  $s_z$  в матрице  $M_s$ :

$$M_s^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Непосредственным умножением легко проверить, что  $M_s \cdot M_s^{-1} = I_{4 \times 4}$ .

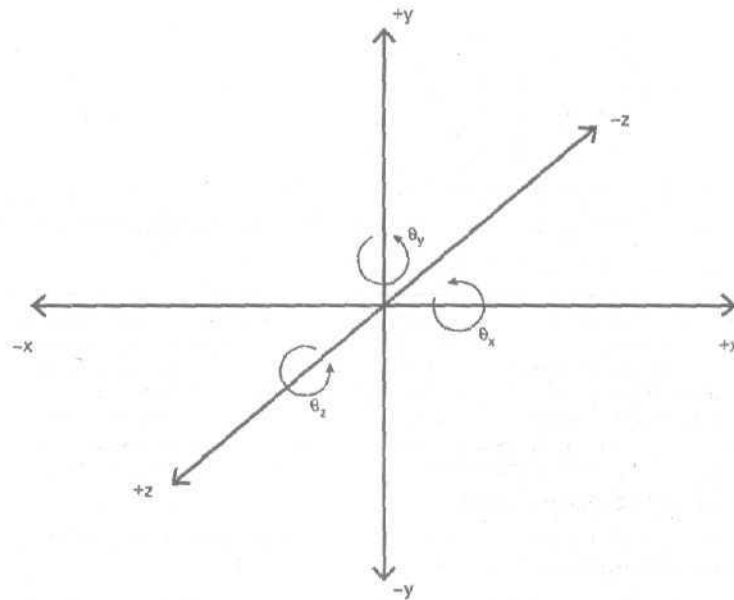
НА ЗАМЕТКУ

Заметим, что вычисление обратной матрицы с использованием метода исключения Гаусса или какого-либо другого — сложная и трудоемкая задача, в то время как здесь мы легко справляемся с ней, исходя из геометрических соображений. Вообще говоря, когда матричное преобразование представляет собой геометрическое преобразование в двумерном или трехмерном пространстве, вычисление обратного матричного преобразования — достаточно тривиальная задача, которая обычно решается путем инверсии геометрического преобразования. Не забывайте об этом, когда мы будем обращаться матрицы поворотов, и попробуйте сначала решить эту задачу самостоятельно, не читая соответствующий подраздел.

## Поворот

Матрица поворота наиболее сложная из всех матриц геометрических преобразований, поскольку она заполнена тригонометрическими функциями. Итак, мы хотим вращать исходную точку с использованием уравнений поворота. Для этого мы должны рассмотреть эти уравнения и разместить в матрице соответствующие операторы. Заметим, что из-за отсутствия переноса в нижней строке матрицы первые три элемента будут нулевыми.

Имеется и **еще** одна техническая деталь — в трехмерном пространстве имеется три оси, вокруг которых может выполняться поворот (рис. 4.28). Начнем с рассмотрения поворота вокруг оси z, а затем используем полученные результаты для создания матриц поворотов вокруг осей x и y. Ключевым свойством поворота вокруг координатной оси является то, что координата точки вдоль этой оси при повороте остается неизменной.



Правая система координат

Рис. 4.28, Оса поворотов

**ВНИМАНИЕ**

Направление поворота для положительных значений угла  $\theta$  зависит от того, используется ли правая или левая система координат. В правой системе координат положительное направление — против часовой стрелки, в левой — по часовой стрелке.

С учетом сказанного вот как выглядят уравнения поворота вокруг осей x, y и z.

Поворот вокруг оси z:

$$M_z = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**НА ЗАМЕТКУ**

Если вы удалите последнюю строку и столбец этой матрицы, то получите матрицу для двумерного поворота в плоскости.

Поворот вокруг оси x описывается следующей матрицей:

$$M_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

И наконец, матрица для поворота вокруг оси  $y$  будет выглядеть таким образом:

$$M_y = \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Теперь рассмотрим конкретный пример поворота вокруг оси  $z$ , чтобы поближе познакомиться с тем, как работает матрица поворота.

Пусть дана точка  $p = [x \ y \ z \ 1]$ , которую мы хотим повернуть на угол  $\theta$  вокруг оси  $z$ :

$$\begin{aligned} p' = p \cdot M_z &= [x \ y \ z \ 1] \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\ &= [(x \cdot \cos \theta - y \cdot \sin \theta) \ (x \cdot \sin \theta + y \cdot \cos \theta) \ z \ 1] \end{aligned}$$

Таким образом, координаты точки после поворота оказываются **следующими**:  $p'((x \cdot \cos \theta - y \cdot \sin \theta), (x \cdot \sin \theta + y \cdot \cos \theta), z)$ . Заметим, что, как и следует, координата  $z$  остается неизменной.

### Обращение поворота

Обращения поворотов вокруг осей  $x$ ,  $y$  и  $z$  вычисляются однотипно, поэтому мы ограничимся обращением поворота вокруг оси  $z$ . Есть два подхода к решению данной задачи: геометрический и основанный на линейной алгебре. Начнем с геометрического подхода. Чтобы вернуть точку в ее начальное положение, ее надо повернуть на тот же угол, но в обратном направлении — т.е. на угол  $-\theta$ . Таким образом, **все**, что нам надо, — это заменить в матрице поворота  $\theta$  на  $-\theta$ :

$$M_z^{-1} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Теперь для упрощения полученной матрицы воспользуемся тождествами четности из табл. 4.5:

$$\sin(-\theta) = -\sin(\theta).$$

$$\cos(-\theta) = \cos(\theta).$$

Применяя эти формулы, можно переписать матрицу обратного поворота следующим образом:

$$M_z^{-1} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Теперь для окончательной проверки корректности найденного решения вычислим произведение  $M_z \cdot M_z^{-1}$  (в процессе вычисления мы воспользуемся теоремой Пифагора, гласящей, что  $\sin^2 \theta + \cos^2 \theta = 1$ ):

$$\begin{aligned}
 M_z \cdot M_z^{-1} &= \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\
 &= \begin{bmatrix} \cos^2 \theta + \sin^2 \theta & -\cos \theta \sin \theta + \sin \theta \cos \theta & 0 & 0 \\ -\sin \theta \cos \theta + \cos \theta \sin \theta & \sin^2 \theta + \cos^2 \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I_{4 \times 4}.
 \end{aligned}$$

Как видите, использование геометрической интерпретации позволяет легко находить обратные матрицы геометрических преобразований. Тем не менее, обратимся к алгебре — дело в том, что матрица поворота интересна еще и тем, что является представителем класса ортонормированных матриц.

### Ортонормированные матрицы

*Ортонормированные матрицы* — это матрицы, в которых каждая строка или столбец ортогональны предыдущей для данного базиса. *Базис* — это набор векторов, который позволяет представить все вектора в пространстве. В нашем случае базис для стандартного двумерного пространства представляет собой  $\langle i + j \rangle$ , а для трехмерного —  $\langle i + j + k \rangle$ . Любой вектор в трехмерном пространстве может быть записан в виде линейной комбинации  $u = u_x \cdot i + u_y \cdot j + u_z \cdot k$ ,

где  $i = \langle 1, 0, 0 \rangle$ ,  $j = \langle 0, 1, 0 \rangle$  и  $k = \langle 0, 0, 1 \rangle$ . Оорты  $i$ ,  $j$  и  $k$  попарно ортогональны, в чем легко убедиться, рассматривая их скалярные произведения. Например,  $i \cdot j = \langle 1, 0, 0 \rangle \cdot \langle 0, 1, 0 \rangle = 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 = 0$ .

Если матрица содержит множество ортогональных векторов в качестве своих столбцов или строк, то такая матрица называется *ортогональной*, а если длины этих векторов равны 1, то ортонормированной. Рассмотрим матрицу поворота вокруг оси  $z$

$$M_z = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Если рассматривать ее строки как вектора, мы получим следующее их множество:

$$\begin{aligned}
 u_1 &= \langle \cos \theta, \sin \theta, 0, 0 \rangle \\
 u_2 &= \langle -\sin \theta, \cos \theta, 0, 0 \rangle \\
 u_3 &= \langle 0, 0, 1, 0 \rangle \\
 u_4 &= \langle 0, 0, 0, 1 \rangle
 \end{aligned}$$

Технически мы имеем дело с четырехмерным пространством, поскольку наши геометрические преобразования основаны на однородных координатах, однако это никак не влияет на ортонормированность матрицы, поскольку скалярные произведения любых двух из приведенных векторов равны 0, в чем очень легко убедиться самостоятельно, Я покажу это только для векторов  $u_1$  и  $u_2$ :

$$u_1 \cdot u_2 = (\cos \theta) \cdot (-\sin \theta) + (\sin \theta)(\cos \theta) + 0 + 0 = 0.$$

Длина всех векторов равна 1, в чем также очень легко убедиться, например вычислив  $u_1 \cdot u_1 = \cos^2 \theta + \sin^2 \theta + 0 + 0 = 1$ .

Теперь вы знаете, что такое ортонормированная матрица и как убедиться в том, что данная матрица является таковой. Все это чудесно, но как это может помочь нам в поисках обратной матрицы? Очень просто: если матрица ортонормированна, то ее транспонирование дает обратную матрицу. Таким образом, в нашем случае  $M_z^T \equiv M_z^{-1}$ . Транспонируя  $M_z$ , мы находим

$$M_z^T = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

которая действительно совпадает с ранее найденной обратной матрицей. Правда, красиво?

## Фундаментальные геометрические объекты

Следующий вопрос, который я хотел рассмотреть, — это фундаментальные геометрические объекты, такие как точки, прямые линии, плоскости и поверхности, но в аспекте различных математических представлений и операций, представляющих интерес в мире трехмерной графики. Например, все многоугольники представляют собой плоскости с границами, так что просто необходимо уметь работать с плоскостями и их уравнениями.

### Точки

Много говорить о точке не получается. На рис. 4.29 изображена пара точек в трехмерном пространстве; если их координаты  $z$  тождественно равны 0, мы получим точки на плоскости. Ясно, что однородные координаты точек включают четвертую координату  $w$ .

### Прямые линии

Прямые линии немного интереснее и несколько сложнее представимы в трехмерном пространстве. Начнем с рассмотрения прямой линии, проходящей через точки  $p_1(x_1, y_1)$  и  $p_2(x_2, y_2)$  на плоскости (рис. 4.30). Такая прямая может быть представлена различными способами — через наклон и точку, наклон и пересечение с осью  $y$  и в общем виде.

#### Уравнение 4.19. Определение прямой при помощи наклона и пересечения с осью $y$

$$y = m \cdot x + b.$$

Здесь  $m$  представляет собой наклон прямой ( $d_y/d_x$ ), а  $b$  — точка пересечения прямой с осью  $y$ .

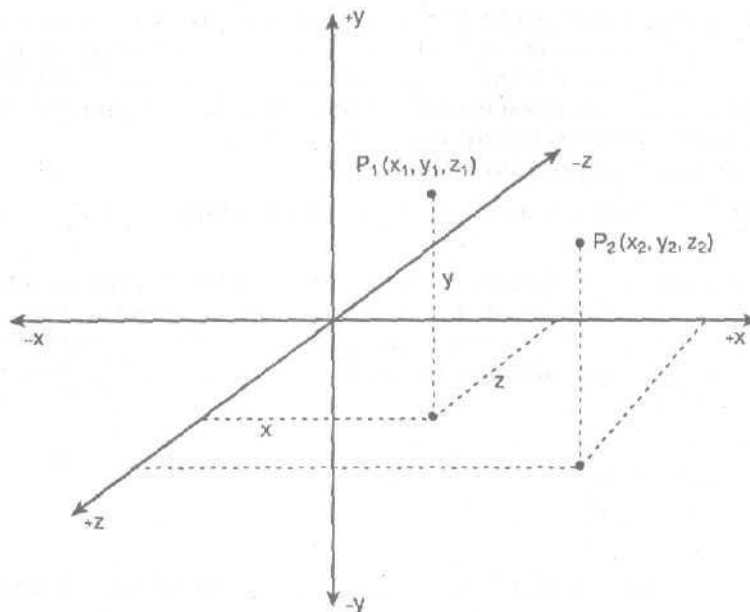


Рис. 4.29. Трехмерные точки

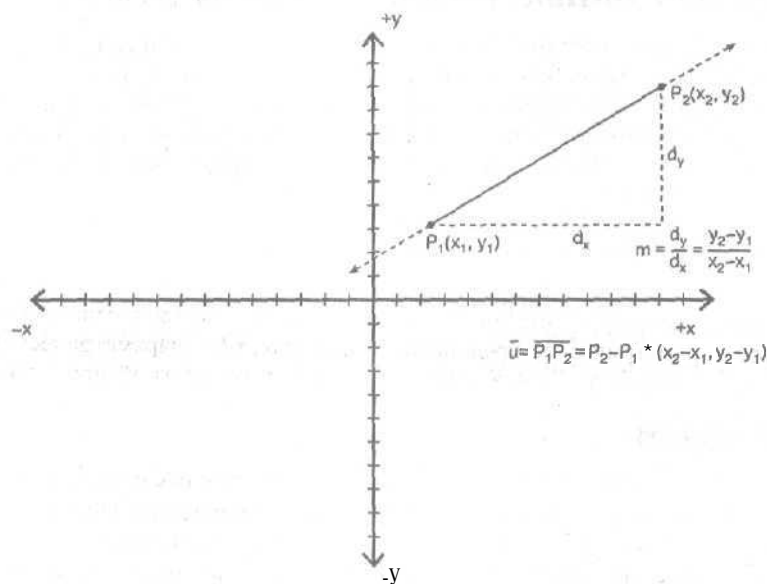


Рис. 4.30. Прямая линия на плоскости

**Уравнение 4.20. Определение прямой при помощи наклона и точки на прямой**

$$(x - x_0) = m \cdot (y - y_0).$$

Здесь  $m$  — наклон прямой, а  $(x_0, y_0)$  — точка на ней.

#### Уравнение 4.21. Общий вид прямой

$$a \cdot x + b \cdot y + c = 0.$$

Естественно, что запись в общем виде наиболее фундаментальна, и в нее легко преобразуются другие представления прямой. Например, преобразование из вида  $y = m \cdot x + b$  очевидно:  $m \cdot x + (-1) \cdot y + b = 0$ . Несколько сложнее записать в общем виде уравнение прямой, проходящей через две точки  $(x_0, y_0)$  и  $(x_1, y_1)$ ? Можно например, вычислить значение наклона  $t$ , подставить его в уравнение 4.20 и привести его к общему виду. Проведите все необходимые вычисления самостоятельно и сравните их с конечным результатом:

$$x \left( \frac{1}{x_1 - x_0} \right) + y \left( \frac{-1}{y_1 - y_0} \right) + \left( \frac{y_0}{y_1 - y_0} - \frac{x_0}{x_1 - x_0} \right) = 0.$$

Для каждого конкретного типа вычислений в большей степени может подходить тот или иной вид уравнения прямой. Однако при вычислении точки пересечения двух прямых лучше всего воспользоваться общим видом, поскольку при этом мы получаем знаковую нам систему уравнений, которую мы уже умеем решать:

$$a_1 \cdot x + b_1 \cdot y = c_1$$

$$a_2 \cdot x + b_2 \cdot y = c_2$$

Можно решить данную систему уравнений, преобразовав ее к матричному виду  $A \cdot X = B$ , где

$$A = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}, B = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}, X = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Как мы уже знаем, решение данного матричного уравнения находится как  $X = A^{-1} \cdot B$ , а обратную матрицу можно вычислить с помощью уравнения 4.17. Я думаю, читатели вполне смогут сделать это самостоятельно.



Если определитель матрицы  $A$  равен 0, единственного решения системы уравнений не существует.

### Прямые в трехмерном пространстве

Представление прямых в трехмерном пространстве — несколько более сложная задача. Кроме того, оно откровенно уродливо! На практике в большинстве случаев будет использоваться параметрическое представление прямых, но о параметрическом представлении я бы хотел поговорить отдельно, в целом, без конкретной привязки к **прямым**. Сейчас же я всего лишь намерен показать, каким образом можно вывести представление трехмерной прямой. Взгляните на рис. 4.31, на котором изображена прямая в трехмерном пространстве, которая проходит через точки  $p_0(x_0, y_0, z_0)$  и  $p_1(x_1, y_1, z_1)$ , и единичный вектор  $\hat{v} = \langle a, b, c \rangle$  от точки  $p_0$  к точке  $p_1$ . Вот как при этом выглядит параметрическое уравнение, определяющее прямую.

#### Уравнение 4.22. Параметрическое задание трехмерной прямой

$$p(x, y, z) = p_0 + \hat{v} \cdot t.$$

При изменении  $t$  от 0 до  $|v|$ , вектор  $p$  проходит от точки  $p_0$  к точке  $p_1$ . Это представление уже должно быть вам знакомо, так как мы сталкивались с его аналогом в двумерном случае.

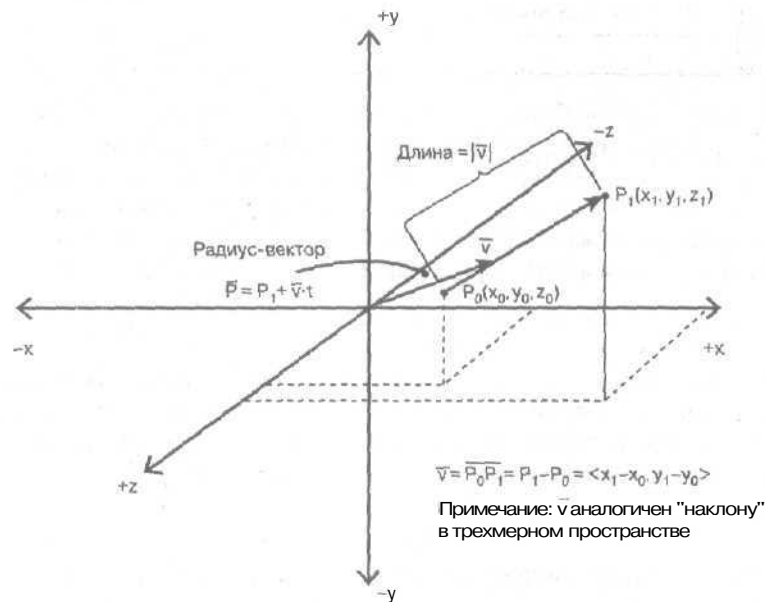


Рис. 4.31. Прямая в трехмерном пространстве

А вот как выглядит явный вид прямой в трехмерном пространстве.

#### Уравнение 4.23. Явное симметричное задание трехмерной прямой

$$\frac{x - x_0}{a} = \frac{y - y_0}{b} = \frac{z - z_0}{c}$$

По сути, *здесь* не одно, а целых три уравнения, но из-за связи их в единое целое вам понадобятся только два из них. Здесь  $a$ ,  $b$  и  $c$  — компоненты единичного вектора  $\hat{v}$ . Основная проблема при работе с таким видом уравнения прямой — в его неуклюжести. Рассмотрим, например, уравнение прямой, проходящей через точки  $P_0(1, 2, 3)$  и  $P_1(5, 6, 7)$ . Тогда

$$v = \langle P_1 - P_0 \rangle = \langle 5 - 1, 6 - 2, 7 - 3 \rangle = \langle 4, 4, 4 \rangle, \text{ и}$$

$$\hat{v} = v / |v| = \langle 4, 4, 4 \rangle / \sqrt{4^2 + 4^2 + 4^2} = \langle 0.577, 0.577, 0.577 \rangle = (a, b, c).$$

Подставляя полученные значения в уравнение 4.23, получаем

$$\frac{x - x_0}{0.577} = \frac{y - y_0}{0.577} = \frac{z - z_0}{0.577}$$

Умножая на 0.577 для устранения знаменателя и разделяя уравнения, получим систему уравнений

$$(x - 1) = (y - 2),$$

$$(y - 2) = (z - 3),$$

или, после упрощения:

$$x - y = -1,$$

$$y - z = -1.$$

Согласитесь, что параметрическое представление выглядит значительно красивее...

## Плоскости

Плоскости в трехмерной графике — главные рабочие лошадки. Технически плоскость представляет собой бесконечный "лист", неограниченно распространяющийся во всех направлениях, как показано на рис. 4.32. Однако при работе с трехмерной графикой мы часто используем концепцию многоугольника, который представляет собой замкнутый объект, лежащий на плоскости и состоящий из множества ребер (рис. 4.33). Поскольку все многоугольники обычно находятся на определенной плоскости, мы можем много сказать о них, обладая информацией о плоскости. Верно и обратное — можно вывести информацию о плоскости, в которой находится многоугольник, и использовать ее в математических операциях.

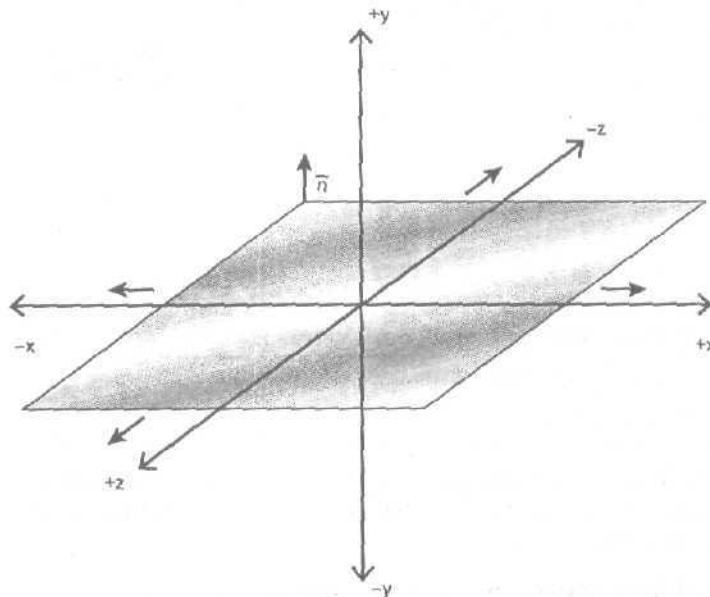


Рис. 4.32. Бесконечная плоскость

Итак, какие же основные свойства плоскостей?

- Все плоскости представляют собой бесконечные "листы" в трехмерном пространстве.
- Плоскости делят пространство на два полупространства (что очень важно для различных алгоритмов разделения пространства и определения столкновений).

Имеется ряд методов для генерации уравнений плоскости. Мы взглянем на проблему с геометрической точки зрения и попробуем вывести уравнение плоскости самостоятельно. Умение выводить что-либо самостоятельно вообще очень помогает в жизни. Впрочем, я отвлеклась. Итак, взгляните на рис. 4.33, где изображена плоскость с нормалью к ней  $\mathbf{n} = \langle a, b, c \rangle$  и точками  $\mathbf{p}_0(x_0, y_0, z_0)$  и  $\mathbf{p}(x, y, z)$  на плоскости. Мы представляем себе эту плоскость, но как записать ее уравнение? Заметим, что поскольку точки  $\mathbf{p}_0$  и  $\mathbf{p}$  лежат на плоскости, вектор  $\overrightarrow{\mathbf{p}_0\mathbf{p}}$  тоже принадлежит плоскости, а значит, он перпендикулярен вектору нормали  $\mathbf{n}$ , какой бы ни была точка  $\mathbf{p}$ . Это уже что-то, так как мы знаем, что если два вектора перпендикулярны, то их скалярное произведение равно 0, т.е.  $\mathbf{n} \cdot \overrightarrow{\mathbf{p}_0\mathbf{p}} = 0$ , а это и есть уравнение плоскости! Конечно, его лучше привести в более понятную форму.

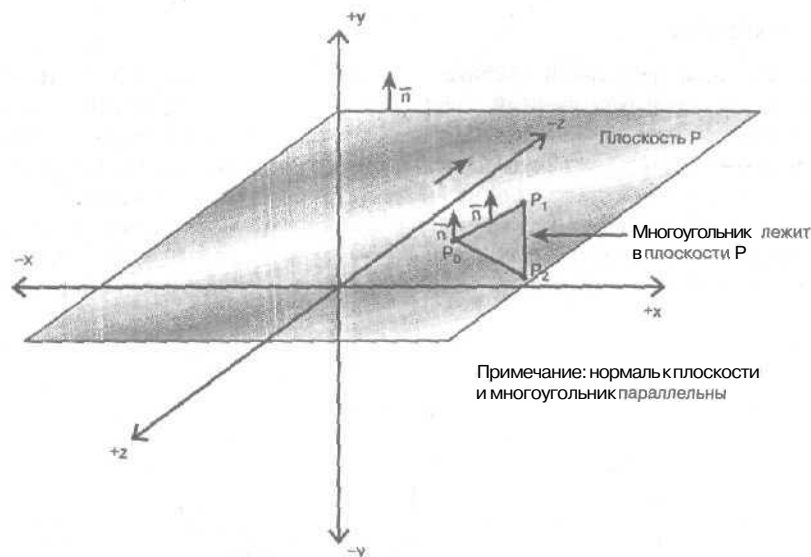


Рис. 4.33. Плоскость, построенная на базе многоугольника

#### Уравнение 4.24. Определение плоскости через точку на ней и вектор нормали

$\langle a, b, c \rangle \cdot (x - x_0, y - y_0, z - z_0) = 0$ , или, выполняя умножение:

$$a \cdot (x - x_0) + b \cdot (y - y_0) + c \cdot (z - z_0) = 0.$$

Это формула плоскости с заданным вектором нормали, проходящей через заданную точку. Если выполнить умножения и собрать все свободные члены вместе, мы получим уравнение плоскости в **общем** виде.

#### Уравнение 4.25. Уравнение плоскости в общем виде

$$a \cdot x + b \cdot y + c \cdot z + d = 0.$$

Уравнение в общем виде удобно в том **случае**, когда требуется вычислить пересечение двух или большего количества плоскостей. В этом случае получается система двух (или иного числа) уравнений относительно  $x$ ,  $y$  и  $z$ ; понятно, что проще всего такая система получается при использовании уравнений в общем виде.

#### Поиск полупространства, содержащего точку

Следует обратить особое внимание на одну достаточно важную операцию, с которой мы будем неоднократно сталкиваться в дальнейшем, — вычисление полупространства, в котором находится определенная точка. Например, вам **потребуется** определить, пересекает ли определенная линия проекции точку или нет, или решить еще какую-то подобную задачу. В этом вам поможет **уравнение** плоскости.

Рассмотрим рис. 4.34, на котором показана **плоскость**, образующиеся благодаря ей **полупространства** и точка  $p$ , для которой следует **определить**, находится ли она в положительном или отрицательной полупространстве. Положительным считается то полупространство, в которое **указывает** вектор **нормали** к плоскости; разумеется, другое полупространство считается отрицательным,

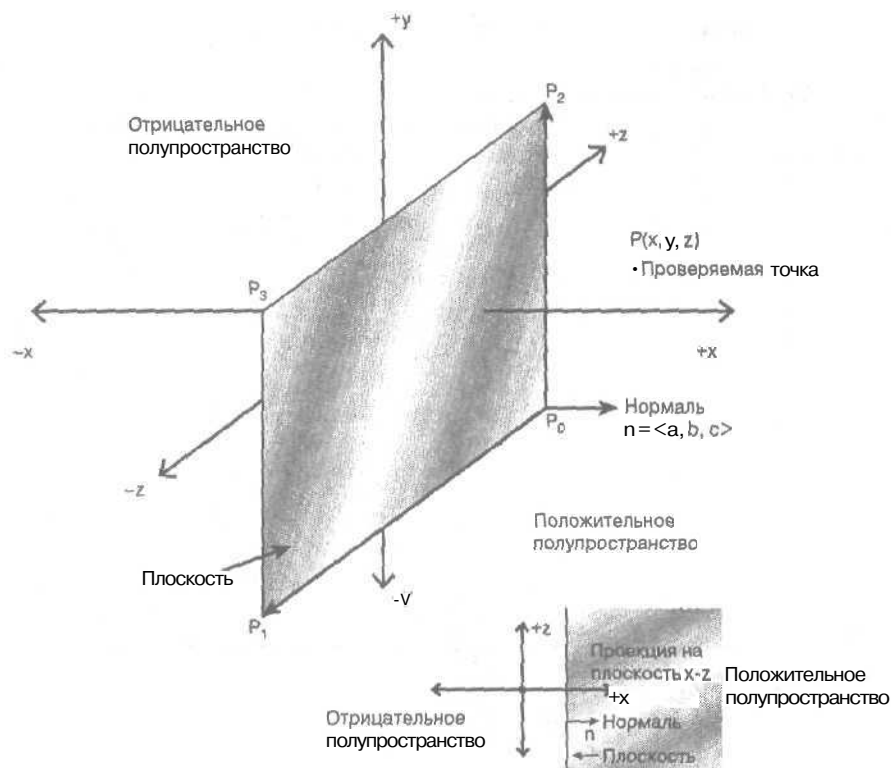


Рис. 4,34. Плоскость и полупространства

Для проверки принадлежности точки тому или иному полупространству предположим, что наше уравнение плоскости записано в следующем виде:

$$hs = a \cdot (x - x_0) + b \cdot (y - y_0) + c \cdot (z - z_0).$$

Все, что нам надо, — это подставить координаты интересующей нас точки в данное уравнение и вычислить значение  $hs$ .

- Если  $hs = 0$ , точка лежит на плоскости.
- Если  $hs > 0$ , точка находится в положительном полупространстве.
- Если  $hs < 0$ , точка находится в отрицательном полупространстве.

Это один из способов проверки принадлежности точки многоугольнику. В качестве конкретного примера рассмотрим плоскость  $xz$ , нормаль к которой равна  $(0, 1, 0)$ , а точка, принадлежащая плоскости, —  $(0, 0, 0)$ . Тогда

$$hs = 0 \cdot (x - 0) + 1 \cdot (y - 0) + 0 \cdot (z - 0) = y.$$

Как видите, координаты  $x$  и  $z$  точки не имеют никакого значения (как и следовало ожидать), и все определяется координатой  $y$  проверяемой точки. Так, например, для точки  $p(10, 10, 10)$ , которая определенно находится в положительном полупространстве, значение  $hs$  равно 10 (положительное значение подтверждает сделанный вывод о размещении точки в положительном полупространстве). Понятно, что в общем случае вам не удастся обойтись проверкой значения одной координаты точки.

## Пересечение плоскости и трехмерной прямой

Рассмотрим очередную задачу, связанную с плоскостью, а именно — задачу поиска пересечения прямой и плоскости в трехмерном пространстве. Это очень важная задача в трехмерных играх, возникающая, например, при определении столкновений, вычислении отсечений и т.п. В качестве примера мы рассмотрим плоскость и прямую, изображенные на рис. 4.35. Для простоты я выбрал плоскость  $xz$ , так что нормаль равна  $\langle 0, 1, 0 \rangle$ , а точка, принадлежащая плоскости, —  $(0, 0, 0)$ . Прямая, пересечение которой с плоскостью нас интересует, проходит через точки  $P_1(4, 4, 4)$  и  $P_2(5, -5, -4)$ . Теперь посмотрим, как нам найти координаты точки пересечения  $P$ .

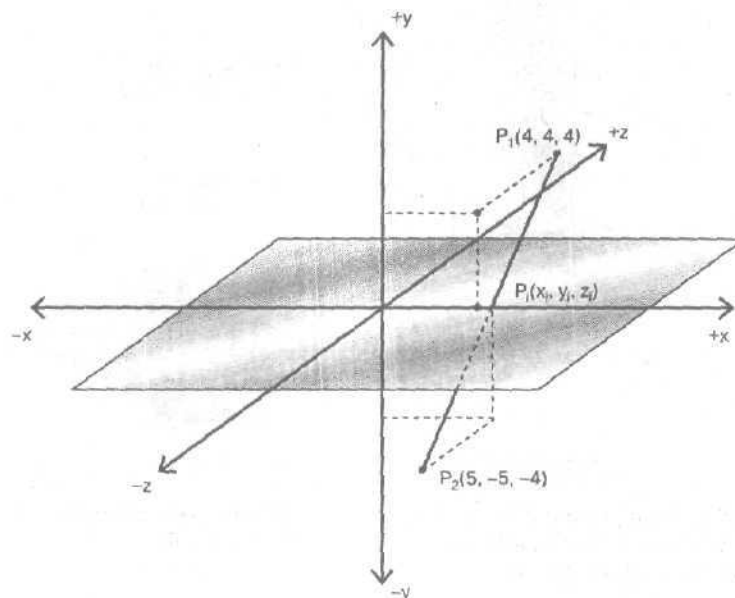


Рис. 4.35. Пересечение прямой и плоскости

Уравнение плоскости, определяемой точкой и вектором нормали, имеет вид:

$$a \cdot (x - x_0) + b \cdot (y - y_0) + c \cdot (z - z_0) = 0.$$

Подставляя сюда вектор нормали и точку  $P_0$ , мы получим уравнение плоскости в виде  $y = 0$ . Уравнение трехмерной прямой в общем виде —

$$\frac{x - x_0}{a} = \frac{y - y_0}{b} = \frac{z - z_0}{c}.$$

Переменные  $\langle a, b, c \rangle$  представляют единичный вектор<sup>2</sup> в направлении прямой, и могут быть вычислены следующим образом:

<sup>2</sup> Заметим, что в силу однородности уравнения прямой здесь может использоваться вектор произвольной длины (мы всегда можем умножить все часта уравнения на одно и то же значение). Таким образом, из описанных далее вычислений можно смело убрать вычисление единичного вектора и в качестве величин  $a$ ,  $b$  и  $c$  использовать компоненты вектора  $v$ , т.е. значения 1, -9 и -8. Вы можете убедиться самостоятельно, что такое изменение никак не повлияет на поиск точки пересечения, но при этом несколько уменьшит трудоемкость ее вычисления. — Прим.ред.

$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_1 = (5, -5, -4) - (4, 4, 4) = (1, -9, -8).$$

Соответственно, единичный вектор вычисляется как

$$\hat{\mathbf{v}} = \mathbf{v}/|\mathbf{v}| = (1, -9, -8)/\sqrt{1^2 + (-9)^2 + (-8)^2} = (0.08, -0.74, -0.66).$$

Подставляя точку  $\mathbf{p}_1$  и вектор  $\hat{\mathbf{v}}$  в уравнение прямой, получаем

$$\frac{x-4}{0.08} = \frac{y-4}{-0.74} = \frac{z-4}{-0.66}.$$

Вторым уравнением в интересующей нас системе является уравнение плоскости, которое, как мы уже выяснили, представляет собой простое равенство  $y = 0$ . Итак, в конечном итоге наша система уравнений имеет следующий вид.

$$(x-4)/0.08 = -(y-4)/0.74,$$

$$-(y-4)/0.74 = -(z-4)/0.66,$$

$$y = 0.$$

Решая эту простейшую систему уравнений, мы находим точку пересечения  $\mathbf{p}_1(4.43, 0, 0.43)$ , которая, судя по рисунку, найдена верно.

## Использование параметрических уравнений

Параметрические уравнения в компьютерной графике гораздо полезнее всех других видов уравнений, поскольку позволяют представлять прямые и кривые линии в виде функций одной переменной. Кроме того, это более естественный способ представления движения и траекторий в компьютерных играх. Поэтому надо не пожалеть времени и старательно изучить параметрические уравнения и работу с ними.

### Двумерные и трехмерные параметрические прямые

Взгляните на рис. 4.36. На нем показаны две точки на плоскости  $xy$ :  $\mathbf{p}_0(x_0, y_0)$  и  $\mathbf{p}_1(x_1, y_1)$ .

Мы знаем, что вектор между этими точками  $\mathbf{v} = \overrightarrow{\mathbf{p}_0\mathbf{p}_1} = (x_1 - x_0, y_1 - y_0)$ , т.е. если мы добавим  $\mathbf{v}$  к  $\mathbf{p}_0$ , то получим точку  $\mathbf{p}_1$ . Вопрос ставится так — каким образом записать это параметрически, с тем, чтобы при изменении некоторого значения  $t$  в некотором закрытом интервале  $[a, b]$  прямая пробегала от точки  $\mathbf{p}_0$  к точке  $\mathbf{p}_1$  вдоль вектора  $\mathbf{v}$ ?

Ответ достаточно прост — множество точек  $(x, y)$  между  $\mathbf{p}_0$  и  $\mathbf{p}_1$  описывается следующим образом.

#### Уравнение 4.26. Уравнение в обобщенном параметрическом виде

$$\mathbf{p} = \mathbf{p}_0 + \mathbf{v} \cdot t, \quad a \leq t \leq b.$$

Единственная проблема при этом заключается в том, что интервал  $t$  в действительности неизвестен. Этот интервал можно вычислить, но здесь есть несколько тонких моментов. Уравнение 4.26 представляет собой обобщенное параметрическое уравнение прямой, которая простирается в бесконечность, так что значение  $t$  может изменяться в пределах от  $-\infty$  до  $+\infty$ , и в этом смысле нам не надо особо беспокоиться о величине  $\mathbf{v}$ . Однако зачастую нам требуется не вся прямая, а конкретный отрезок, а здесь главное отличие в том, что интервал значений параметра должен быть точно определен — и здесь есть два основных способа определения значения  $\mathbf{v}$ , соответственно, интервала значений  $t$ .

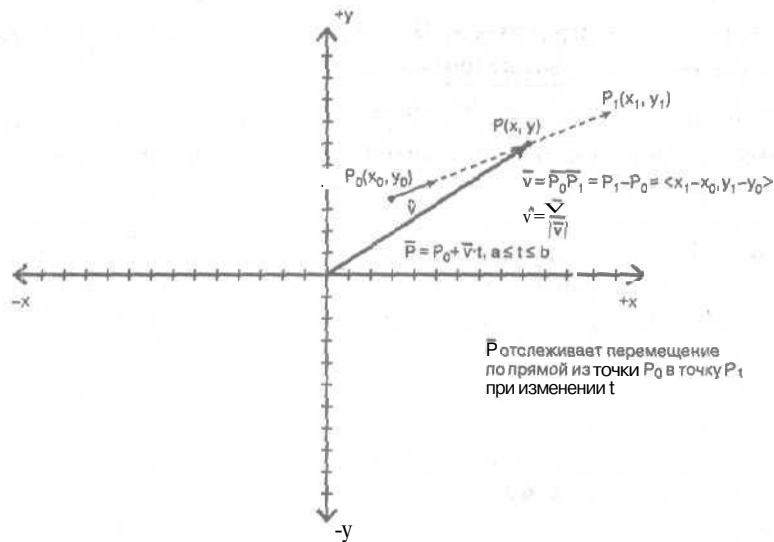


Рис. 4.36. Параметрическое представление прямой

## Параметрическое представление отрезка с помощью стандартного вектора направления

В предыдущем разделе мы определили  $\vec{v} = \overrightarrow{P_0 P_1} = \langle x_1 - x_0, y_1 - y_0 \rangle$ ; таким образом, длина вектора  $\vec{v}$  в точности равна расстоянию от точки  $P_0$  до точки  $P_1$ , как показано на рис. 4.36. Поэтому вполне очевидно, что значения  $t$  при этом должны находиться в интервале  $[0, 1]$ .

### НА ЗАМЕТКУ

Запись интервала с использованием квадратных скобок означает закрытый интервал, т.е. включающий конечные точки, в то время как круглые скобки означают открытый интервал, в который указанные конечные точки не входят. Так, запись  $(0, 1]$  означает все действительные числа от 0 до 1, включая 1, но исключая 0.

Убедимся в этом, рассматривая конечные точки интервала. При  $t=0$  формула  $P = P_0 + \vec{v} \cdot t$  превращается в  $P = P_0$ , как и ожидалось. При  $t=1$  мы получаем  $P = P_0 + \vec{v} \cdot 1 = P_0 + (P_1 - P_0) = P_1$ , что также совершенно правильно. Следовательно, интервал  $[0, 1]$  описывает интересный нас отрезок от точки  $P_0$  до точки  $P_1$ . Мы можем использовать и другие значения  $t$ , но при этом получающиеся точки будут лежать на прямой, проходящей через точки  $P_0$  и  $P_1$ , но вне пределов указанного отрезка. Это очень важное свойство, используемое при решении параметрических систем уравнений. Если получаемое при этом значение  $t$  выходит за рамки интервала, мы знаем, что полученная в результате точка находится вне отрезка, что может оказаться крайне важной деталью при решении задач трехмерной геометрии.

Использование интервала  $[0, 1]$  очень удобно с разных точек зрения, но давайте рассмотрим еще одно представление — с использованием нормализованного вектора  $\hat{v}$ .

## Параметрическое представление отрезка с помощью единичного вектора направления

Следующий вариант параметрического представления прямой аналогичен предыдущему, но с одним небольшим изменением: вместо вектора  $v$  мы используем нормализованный вектор  $\hat{v}$ :  $p = p_0 + \hat{v} \cdot t$ . Здесь  $\hat{v} = v/|v|$ , где, как и ранее,  $v = p_0 p_1 = \langle x_1 - x_0, y_1 - y_0 \rangle$ .

При этом встает вопрос о том, какой же интервал значений  $t$  определяет отрезок между точками  $p_0$  и  $p_1$ , как показано на рис. 4.37.

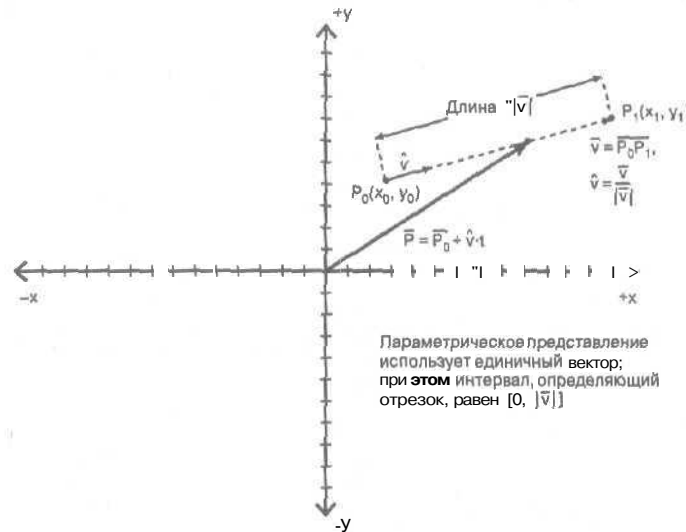


Рис. 4.37. Интервал, определяющий отрезок между точками  $p_0$  и  $p_1$

Если вы внимательно изучили предыдущий материал, то можете дать верный ответ не задумываясь; это интервал  $[0, |v|]$ . Убедитесь сами.

При  $t=0$  имеем  $p = p_0 + \hat{v} \cdot 0 = p_0$ . Когда  $t = |v|$ , мы получаем

$$p = p_0 + \hat{v} \cdot |v| = p_0 + \frac{v}{|v|} \cdot |v| = p_0 + v = p_1,$$

т.е. именно то, что и требовалось. В конечном счете между этими двумя представлениями нет особой разницы. Аналогично можно разработать множество вариантов параметрических представлений — например, представление, когда отрезку соответствует интервал  $[-1, 1]$ . Попробуйте вывести такую формулу самостоятельно.

## Параметрическое представление трехмерных прямых

Трехмерное параметрическое представление прямых абсолютно идентично двумерной версии, надо только учесть, что теперь и точки, и вектор  $v$  имеют по три компонента. Векторное же уравнение остается тем же;  $p = p_0 + v \cdot t$ , где  $v = p_1 - p_0$ .

Обратите внимание, насколько прост оказался переход от двумерного к трехмерному пространству и насколько он был сложен при явном использовании координат для представления прямых. Вот почему работа с прямыми в трехмерном игровом процессоре

осуществляется с использованием параметрического представления — в явном виде очень сложно осуществить все необходимые вычисления.

## Вычисление пересечения параметризованных прямых

Рассмотрим задачу движения двух кораблей по плоскости, показанную на рис. 4.38. Как видно из рисунка:

- корабль 1 движется из точки  $P_0$  в точку  $P_1$ ;
- # корабль 2 движется из точки  $P_2$  в точку  $P_3$ .

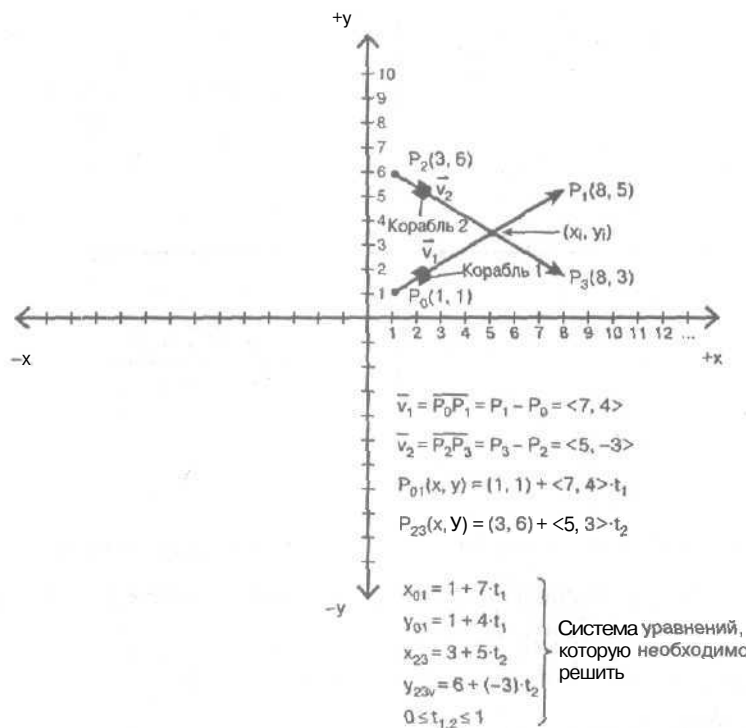


Рис. 4.38. Движение кораблей по плоскости

Мы хотим определить, пересекутся ли их траектории. Заметьте, мы ничего не говорим о том, столкнутся ли сами **корабли**, поскольку время не входит в нашу задачу. Пока что нас интересует только возможность пересечения их путей. Первое **решение**, которое приходит на ум, состоит в том, чтобы построить соответствующие прямые и найти точку их пересечения — но как после этого вы определите, пересекаются ли **интересующие** нас **отрезки**? Вот почему так удобно параметрическое представление — в этом случае мы определяем значения параметров  $t$  в точке пересечения. Однако если  $t$  находится за пределами интервала, определяющего отрезок, мы знаем, что отрезки не пересекаются (рис. 4.39).

Начнем с построения двух параметрических прямых с параметрами в интервале  $[0, 1]$  для простоты решения. Заметим, что если интервалы будут разными, то мы окажемся в глупой ситуации сравнения литров с метрами.

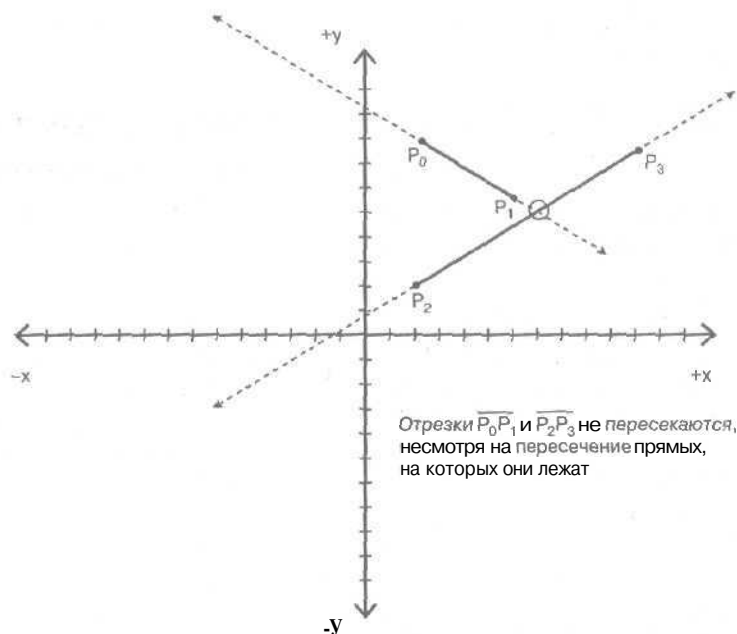


Рис. 4.39. Пересечение прямых не означает пересечение отрезков

Итак, приступим к математической записи задачи.

- Корабль 1 движется из точки  $p_0(1,1)$  в точку  $p_1(8,5)$ , вектор направления  $v_1 = \langle 8-1, 5-1 \rangle = \langle 7, 4 \rangle$ .
- Корабль 2 движется из точки  $p_2(3,6)$  в точку  $p_3(8,3)$ , вектор направления  $v_2 = \langle 8-3, 3-6 \rangle = \langle 5, -3 \rangle$ .

В результате параметрические уравнения отрезков имеют вид ( $0 < t_1, t_2 < 1$ ).

$$p_{01} = (1,1) + \langle 7,4 \rangle \cdot t_1,$$

$$p_{23} = (3,6) + \langle 5,-3 \rangle \cdot t_2.$$

Обратите внимание на наличие разных параметров  $t$  для каждой прямой. Каждый из этих параметров находится в интервале  $[0,1]$ , но они не обязаны быть одинаковы в точке пересечения отрезков (если таковая имеется). Следовательно, мы должны работать с двумя разными параметрами, и забыть об этом — значит получить массу проблем.

Запишем полученные уравнения для каждой компоненты по отдельности.

$$x = 1 + 7 \cdot t_1,$$

$$y = 1 + 4 \cdot t_1,$$

$$x = 3 + 5 \cdot t_2,$$

$$y = 6 - 3 \cdot t_2.$$

Теперь мы должны приравнять значения  $x$  и  $y$  для каждой прямой (поскольку в точке пересечения они совпадают), решить полученную систему уравнений относительно параметров  $t_1$  и  $t_2$  и убедиться, что параметры находятся в интервале  $[0,1]$ . Если это так, отрезки пересекаются. Приравняв значения  $x$  и  $y$  для каждой прямой, получаем следующее.

$$x: 1 + 7 \cdot t_1 = 3 + 5 \cdot t_2,$$

$$y: 1 + 4 \cdot t_1 = 6 - 3 \cdot t_2.$$

Проводя тривиальные преобразования, получим систему уравнений

$$7 \cdot t_1 - 5 \cdot t_2 = 2,$$

$$4 \cdot t_1 + 3 \cdot t_2 = 5,$$

решая которую каким угодно способом — по правилу Крамера, методом исключения Гаусса или при помощи колдовства, — мы получим окончательный ответ  $t_1 = 0.756$  и  $t_2 = 0.658$ .

Поскольку оба найденных значения параметров  $t_1$  и  $t_2$  находятся в интервале  $[0, 1]$ , мы делаем вывод, что отрезки пересекаются. Если мы хотим выяснить, в какой именно точке они пересекаются, нам надо подставить полученные значения  $t_1$  и  $t_2$  в уравнения отрезков. Подставим их в оба уравнения, чтобы не только найти точку пересечения отрезков, но и убедиться в правильности нашего решения (координаты точки пересечения, полученные из уравнения первого и второго отрезков, должны совпадать).

Для отрезка 1:

$$x = 1 + 7 \cdot t_1 = 6.29,$$

$$y = 1 + 4 \cdot t_1 = 4.02.$$

Для отрезка 2:

$$x = 3 + 5 \cdot t_2 = 6.29,$$

$$y = 6 - 3 \cdot t_2 = 4.02.$$

Как видите, координаты точки пересечения, вычисленные из двух **уравнений** прямых, совпадают, что служит доказательством корректности нашего решения поставленной задачи. Надеюсь, это решение окончательно убедило вас в **преимуществах** использования параметрического уравнения прямой.

Решение задачи пересечения отрезков в трехмерном случае совершенно идентично только что рассмотренному. При этом не появляются новые уравнения, но появляется новая переменная  $z$ . Уравнения отрезков изменяются **следующим** образом:

$$P_{01}(x, y, z) = (x_0, y_0, z_0) + \langle v_{x1}, v_{y1}, v_{z1} \rangle \cdot t_1,$$

$$P_{23}(x, y, z) = (x_2, y_2, z_2) + \langle v_{x2}, v_{y2}, v_{z2} \rangle \cdot t_2.$$

Расписав их покомпонентно и приравняв значения  $x$ ,  $y$  и  $z$ , получим три уравнения для поиска двух параметров  $t_1$  и  $t_2$ :

$$x_0 + v_{x1} \cdot t_1 = x_2 + v_{x2} \cdot t_2,$$

$$y_0 + v_{y1} \cdot t_1 = y_2 + v_{y2} \cdot t_2,$$

$$z_0 + v_{z1} \cdot t_1 = z_2 + v_{z2} \cdot t_2.$$

Обратите внимание на то, что неизвестных переменных две, а уравнений — три. В этом случае можно поступить следующим образом — найти  $t_1$  и  $t_2$  из первых двух уравнений и подставить в третье, чтобы убедиться, что оно также удовлетворяется. Если подстановка  $t_1$  и  $t_2$  в третье уравнение дает равенство, это означает, что прямые пересекаются, и остается выяснить, пересекаются ли отрезки — путем проверки принадлежности  $t_1$  и  $t_2$  интервалу  $[0, 1]$  - Если же третье уравнение не превращается в равенство, это означает, что прямые не пересекаются. **Вообще** говоря, пересечение прямых — большая редкость в трехмерном пространстве, в отличие от пересечения прямой и плоскости, чем мы сейчас и займемся.

## Вычисление пересечения отрезка и плоскости

Вычисление пересечения отрезка и плоскости **может** быть как очень простым, так и очень сложным — в зависимости от плоскости. Если это плоскость  $xy$ ,  $xz$ ,  $yz$  или параллельная одной из них, вычисления становятся тривиальными. Однако в общем случае задача усложняется.

Начнем с записи формул отрезка и плоскости и решения в тривиальном случае. Уравнение отрезка имеет вид  $\mathbf{p} = \mathbf{p}_0 + \mathbf{v} \cdot t$ ,  $0 \leq t \leq 1$ , или, расписывая более детально —  $p(x, y, z) = p_0(x_0, y_0, z_0) + \{v_x, v_y, v_z\} \cdot t$ . Уравнение плоскости, как вы помните —  $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}'_0) = 0$ , или более подробно —  $n_x \cdot (x - x'_0) + n_y \cdot (y - y'_0) + n_z \cdot (z - z'_0) = 0$ . Штрих у точки  $\mathbf{p}'_0$  поставлен для того, чтобы вы случайно не перепутали эти точки — начальный конец отрезка и точку, принадлежащую плоскости.

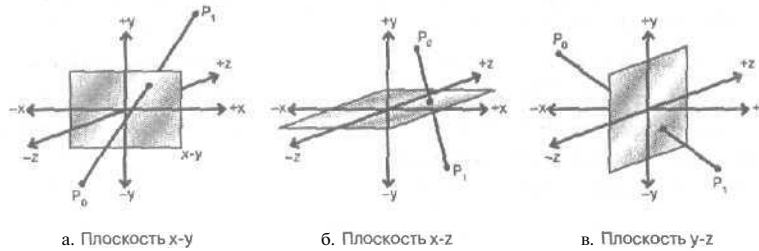


Рис. 4.40. Пересечение отрезка и плоскости (тривиальные случаи)

На рис. 4.40 показаны интересующие нас в настоящий момент тривиальные случаи пересечения плоскости и отрезка. Все решения тривиальны и практически **идентичны**, поэтому мы рассмотрим в качестве примера только пересечение отрезка с плоскостью  $xy$ . Мы знаем, что в этом случае точка пересечения имеет нулевую  $z$ -координату, так что нам очень легко получить значение  $t$  из уравнения  $z = z_0 + v_z \cdot t$ . Подставляя  $z = 0$ , получаем  $t = -z_0/v_z$ . Если это значение находится в интервале  $[0, 1]$ , значит, отрезок пересекается с плоскостью, а координаты точки пересечения легко найти, подставляя вычисленное значение  $t$  в остальные уравнения:

$$x = x_0 + v_x \cdot (-z_0/v_z),$$

$$y = y_0 + v_y \cdot (-z_0/v_z).$$

Решение в общем виде немного сложнее. Произвольная плоскость имеет уравнение

$$\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}'_0) = 0,$$

$$n_x \cdot (x - x'_0) + n_y \cdot (y - y'_0) + n_z \cdot (z - z'_0) = 0,$$

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + (-n_x \cdot x'_0 - n_y \cdot y'_0 - n_z \cdot z'_0) = 0.$$

В общем виде это уравнение записывается как  $a \cdot x + b \cdot y + c \cdot z + d = 0$ , где  $a = n_x$ ,  $b = n_y$ ,  $c = n_z$ , а  $d = (-n_x \cdot x'_0 - n_y \cdot y'_0 - n_z \cdot z'_0)$ .

Уравнение отрезка можно записать в виде следующей системы уравнений:

$$x = x_0 + v_x \cdot t,$$

$$y = y_0 + v_y \cdot t,$$

$$z = z_0 + v_z \cdot t.$$

Теперь вам уже все ясно? Если мы подставим правые части этой системы уравнений в уравнение **плоскости**, то мы получим уравнение с одним неизвестным  $t$ . Найдя  $t$  (уравнение 4.27), мы определяем, имеется ли пересечение отрезка с плоскостью или нет, и если да, то подстановкой полученного значения  $t$  в уравнения отрезка мы найдем координаты точки пересечения.

#### Уравнение 4.27. Пересечение отрезка и плоскости

$$t = -(a \cdot x_0 + b \cdot y_0 + c \cdot z_0 + d) / (a \cdot v_x + b \cdot v_y + c \cdot v_z).$$

Маленькое предупреждение — перед выполнением деления не забудьте убедиться, что знаменатель не равен нулю!

НА ЗАМЕТКУ

Данное алгебраическое решение корректно, но есть и более красивое геометрическое решение с использованием векторов. Попробуйте найти его сами. Я только напомним, что и числитель, и знаменатель в уравнении 4.27 представляют собой скалярные произведения векторов.

Теперь вы знаете вполне достаточно для того, чтобы с успехом решать трехмерные задачи определения столкновений, отсечения, траекторий стрельбы и множество других. Мы еще встретимся со всеми этими задачами далее в книге, так что не растеряйте приобретенные знания до того времени.

## Введение в кватернионы

Программирование современных трехмерных игр сложно представить себе без **кватернионов**, которые были открыты в 19 веке Вильямом Гамильтоном (William Rowan Hamilton) (если вы изучали теорию графов, то должны быть знакомы с **Гамильтоновыми** цепями). Кватернионы не были разработаны специально для трехмерной графики (это очевидно), но нашли в этой области широкое применение.

Кватернионы основаны на комплексных числах, которые представляют собой немного сложную для понимания абстракцию и не имеют физического смысла в реальном мире — они имеют только математический смысл. Однако при этом комплексные числа являются инструментом, который можно использовать для представления математических идей там, где действительных чисел недостаточно. Давайте начнем с азов теории комплексных чисел, после чего перейдем к кватернионам и их математическим свойствам, а затем рассмотрим, каким образом они могут применяться в компьютерной графике и играх.

## Теория комплексных чисел

Множество действительных чисел  $K$  состоит из всех чисел в интервале  $(-\infty, +\infty)$ . Все просто и понятно, но вот три уравнения

$$x = \sqrt{4} = 2,$$

$$x = \sqrt{1} = 1,$$

$$x = \sqrt{-1} = ?$$

Третье уравнение представляет собой проблему — такого действительного числа, которое, будучи возведено в квадрат, даст нам  $-1$ , не существует. Нам надо ввести новое число, которое мы назовем **мнимой единицей**  $i$  ( $j$  в электротехнике) и которое обладает следующим **свойст-**

ВМ:  $i = \sqrt{-1}$ . Тогда  $i \cdot i = -1$  и, соответственно, мы можем находить корни из отрицательных чисел. Так, например,  $\sqrt{-4} = 2 \cdot i$ , поскольку  $(2 \cdot i) \cdot (2 \cdot i) = 4 \cdot i^2 = 4 \cdot -1 = -4$ .

Работая с мнимыми числами, вы можете рассматривать мнимую единицу как обычную переменную или коэффициент. Все правила обычной алгебры остаются в силе. Таким образом, все математические действия выполняются, как обычно; однако после их выполнения следует найти все члены  $i$  и заменить их на  $-1$  и упростить полученное выражение. Рассмотрим, например, преобразования суммы

$$\begin{aligned} 3 + 5 \cdot i + 3 \cdot i^2 - 10 + 6 \cdot i &= \\ &= 3 \cdot i^2 + i \cdot (5 + 6) + (3 - 10) = \\ &= 3 \cdot (-1) + 11 \cdot i - 7 = \\ &= -10 + 11 \cdot i. \end{aligned}$$

Ничего экстраординарного. Однако сами по себе мнимые числа не так уж полезны, так что математики предпочитают работать с *комплексными числами*, которые представляют собой сумму действительного и мнимого числа, т.е. числа следующего вида:  $z = a + b \cdot i$ . Величина  $a$  называется действительной частью числа, а  $b$  — мнимой частью. Поскольку  $a$  и  $b$  не могут быть суммированы в силу наличия коэффициента  $i$ , можно рассматривать комплексные числа как точки на комплексной *плоскости*, показанной на рис. 4.41.

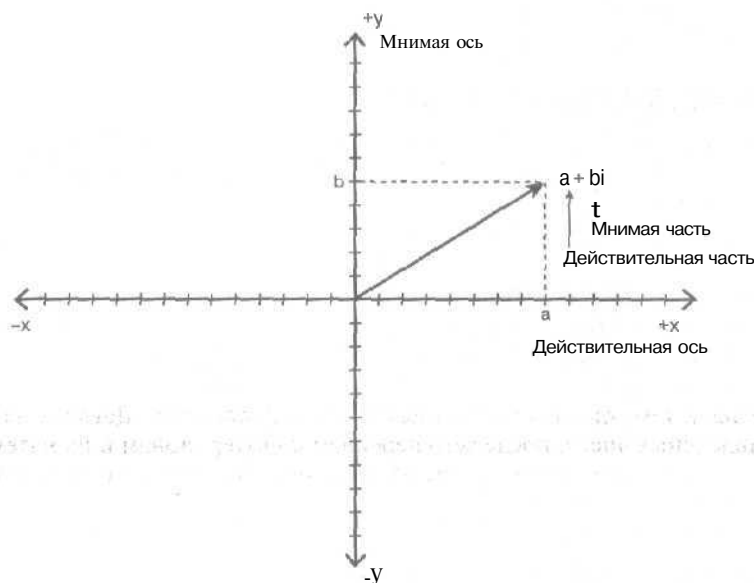


Рис. 4.41. Комплексная плоскость

По соглашению, ось  $x$  представляет собой ось действительных чисел, а ось  $y$  — ось мнимых чисел. Таким образом, у нас есть геометрическая интерпретация комплексных чисел на векторном базисе:  $z = a \cdot (1, 0) + b \cdot (0, i)$ . Я еще вернусь к этой концепции позже, а пока мы познакомимся со сложением, умножением и другими операциями с комплексными числами.

### Умножение и деление комплексных чисел на скаляр

Умножение комплексного числа на скаляр выполняется на основе умножений компонентов *следующим* образом.

$$z = (a + b \cdot i),$$

$$k \cdot z = k \cdot (a + b \cdot i) = k \cdot a + (k \cdot b) \cdot i.$$

$$\text{Например: } 3 \cdot (2 + 5 \cdot i) = 6 + 15 \cdot i.$$

Деление комплексного числа на скаляр выполняется аналогично, поскольку деление можно заменить умножением на обратное число.

### Комплексное сложение и вычитание

Для сложения и вычитания комплексных чисел надо просто сложить (или вычесть) действительные и мнимые части по отдельности.

$$z_1 = a + b \cdot i,$$

$$z_2 = c + d \cdot i,$$

$$z_1 + z_2 = (a + c) + (b + d) \cdot i.$$

$$\text{Например, } (3 + 5 \cdot i) + (12 - 2 \cdot i) = 15 + 3 \cdot i.$$

### Аддитивный нулевой элемент

Аддитивный нулевой элемент представляет собой комплексное число, которое, будучи добавленным к другому комплексному числу, не изменяет его значения. Таким числом, само собой разумеется, является  $0 + 0 \cdot i$ , так как

$$(0 + 0 \cdot i) + (a + b \cdot i) = (a + 0) + (b + 0) \cdot i = a + b \cdot i.$$

### Аддитивный обратный элемент

Комплексным аддитивным обратным элементом называется такое число, которое, будучи прибавленным к данному, дает ноль. Совершенно очевидно, что таковым для числа  $z$  является число  $-z$ .

$$(a + b \cdot i) + (-a - b \cdot i) = (a - a) + (b - b) \cdot i = 0 + 0 \cdot i.$$

### Комплексное умножение

Умножение комплексных чисел ненамного сложнее сложения. Его очень легко выполнить, считая, что  $i$  — просто некоторая переменная.

$$z_1 = a + b \cdot i,$$

$$z_2 = c + d \cdot i,$$

$$z_1 \cdot z_2 = (a + b \cdot i) \cdot (c + d \cdot i) =$$

$$= a \cdot c + a \cdot d \cdot i + b \cdot c \cdot i + b \cdot d \cdot i^2 =$$

$$= (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c) \cdot i.$$

$$\text{Например, } (1 + 2 \cdot i) \cdot (-3 + 3 \cdot i) = (-3 - 6) + (-6 + 3) \cdot i = -9 - 3 \cdot i.$$

### Комплексное деление

Деление комплексных чисел можно выполнить грубо "в лоб". Например, мы можем вычислить частное двух комплексных чисел следующим образом:

$$z_1 = a + b \cdot i,$$

$$z_2 = c + d \cdot i,$$

$$\frac{z_1}{z_2} = \frac{a + b \cdot i}{c + d \cdot i}.$$

Если  $c$  или  $d$  равно 0, деление становится тривиальным, но если и  $c$ , и  $d$  имеют ненулевые значения, придется немного потрудиться. Вопрос в том, как вновь привести полученную дробь к виду  $a + b \cdot i$ . Здесь нам поможет понятие комплексного сопряженного числа (обычно обозначается звездочкой), которое имеет ту же действительную часть, что и исходное число, и мнимую часть, равную мнимой части исходного числа с обратным знаком:

$$z = a + b \cdot i,$$

$$z^* = a - b \cdot i,$$

Сопряженные числа обладают тем свойством, что произведение комплексного числа на сопряженное дает в результате действительное число:

$$z \cdot z^* = (a + b \cdot i) \cdot (a - b \cdot i) = a^2 + b^2.$$

Правда, красиво? Теперь мы можем умножить и числитель, и знаменатель нашей дроби на комплексное сопряженное знаменателю число и привести ее таким образом к желаемому виду:

$$\begin{aligned} \frac{a + b \cdot i}{c + d \cdot i} &= \frac{a + b \cdot i}{c + d \cdot i} \cdot \frac{c - d \cdot i}{c - d \cdot i} = \frac{(a + b \cdot i) \cdot (c - d \cdot i)}{c^2 + d^2} = \\ &= \frac{a \cdot c + b \cdot c \cdot i - a \cdot d \cdot i + b \cdot d \cdot a}{c^2 + d^2} = \frac{a \cdot c + b \cdot d}{c^2 + d^2} + \frac{b \cdot c - a \cdot d}{c^2 + d^2} \cdot i. \end{aligned}$$

Может, выглядит и несколько громоздко, но зато частное двух комплексных чисел приведено к желаемому виду суммы действительной и мнимой частей  $a + b \cdot i$ .

### Мультипликативный обратный элемент

Последнее математическое свойство, необходимое для того, чтобы множество комплексных чисел было замкнутым, — наличие мультипликативного обратного элемента, т.е. комплексного числа, которое, будучи умноженным на данное, даст в результате 1. После небольшого анализа становится очевидным способ поиска мультипликативного обратного: надо просто разделить 1 на исходное комплексное число при помощи обычной операции деления и привести полученное значение к стандартному виду комплексного числа умножением числителя и знаменателя на комплексное сопряженное знаменателя:

$$z = a + b \cdot i,$$

$$\frac{1}{z} = \frac{1}{a + b \cdot i} = \frac{1}{a + b \cdot i} \cdot \frac{a - b \cdot i}{a - b \cdot i} = \frac{a - b \cdot i}{a^2 + b^2} = \frac{a}{a^2 + b^2} - \frac{b}{a^2 + b^2} \cdot i.$$

Нетрудно убедиться, что умножение полученного числа на исходное дает, как и требовалось,  $1 + 0 \cdot i$ .

### Комплексные числа как векторы

Сейчас я хочу еще раз обратиться к представлению комплексных чисел в виде векторов в двумерной плоскости. Обратитесь еще раз к рис. 4.41, где комплексное число представлено в виде точки в декартовых координатах ( $x$ -координата точки равна действительной части комплексного числа, а  $y$ -координата — мнимой). Таким образом, комплексное число  $z = a + b \cdot i$  можно представить в виде вектора  $z = a \cdot (1, 0) + b \cdot (0, i)$  или, более компактно,  $z = \langle a, b \rangle$ . На рис. 4.42 показано рассмотренное нами представление комплексного числа в виде вектора.

Такое представление позволяет нам преобразовывать комплексное число как вектор, получая при этом совершенно корректные результаты. Кроме того, такое представление обеспечивает визуализацию абстрактной математической записи комплексных чисел, облегчая понимание соотношений, которые иначе можно представить только в чисто математической форме.

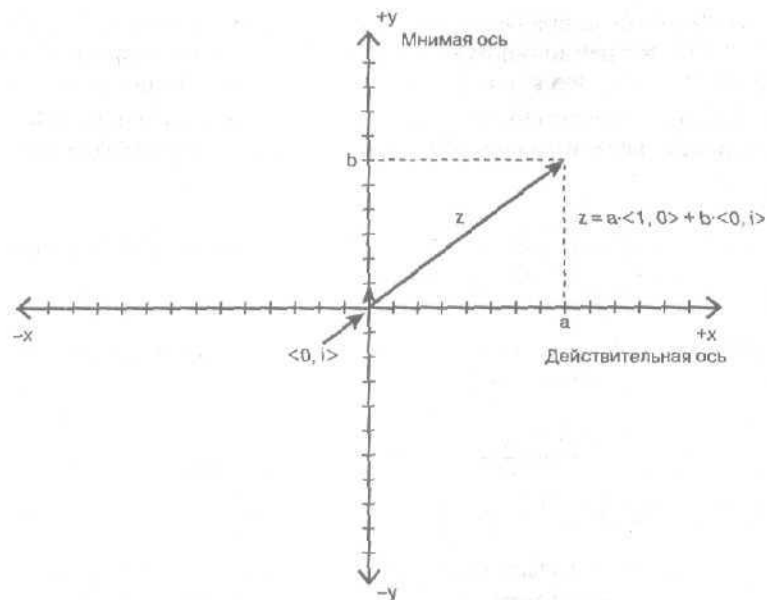


Рис. 4.42. Представление комплексных чисел в виде векторов

### Норма комплексного числа

Зачастую возникает задача поиска "длины" комплексного числа. Ясно, что с точки зрения чистой математики, особого смысла в этом нет, но при использовании представления комплексного числа в виде вектора понятие длины приобретает простой и понятный смысл. Длина, или *норма* комплексного числа, вычисляется следующим образом.

#### Уравнение 4.28. Норма комплексного числа

$$z = a + b \cdot i,$$

$$|z| = \sqrt{a^2 + b^2}.$$

Заметим также, что если воспользоваться комплексно сопряженным числом, то норму можно записать следующим эквивалентным образом:

$$|z| = \sqrt{z \cdot \bar{z}}.$$

### Гиперкомплексные числа

Кватернионы представляют собой не что иное, как *гиперкомплексные* числа. Термин "гиперкомплексные" означает, что это комплексные числа с более чем одной мнимой компонентой. В случае кватернионов имеется одна действительная часть и три мнимых.

Кватернионы можно записать многими различными способами, но обычно в общем виде они записываются следующим образом.

#### Уравнение 4.29. Запись кватернионов

$$q = q_0 + q_1 \cdot i + q_2 \cdot j + q_3 \cdot k,$$

или

$$q = q_0 + q_v, \text{ где } q_v = q_1 \cdot i + q_2 \cdot j + q_3 \cdot k.$$

Здесь  $q_i$  — действительные числа, а  $i, j$  и  $k$  — мнимые числа, которые образуют векторный базис кватерниона. Значение  $q_0$  — действительное, не имеющее мнимого коэффициента.

Мнимый базис  $\langle i, j, k \rangle$  имеет ряд интересных свойств. Его можно рассматривать как множество трехмерных взаимно перпендикулярных единичных векторов в мнимой системе координат, как показано на рис. 4.43. Мнимый базис обладает следующим интересным свойством.

#### Уравнение 4.30. Произведения элементов базиса кватернионов

$$i^2 = j^2 = k^2 = -1 = i \cdot j \cdot k.$$

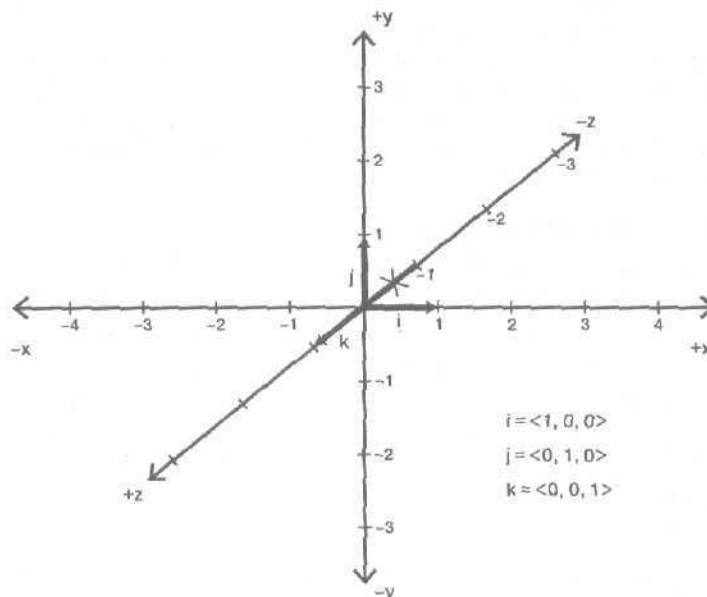


Рис. 4.43. Гиперкомплексное трехмерное пространство

$\sum_{i=0}^{\infty}$

Элементы мнимого базиса  $\langle i, j, k \rangle$  выделены полужирным шрифтом, поскольку в силу их дуальной природы их можно рассматривать и как переменные, и как векторы.

Часть " $= i \cdot j \cdot k$ ", наверное, вызывает определенное удивление, но она совершенно корректна. Более того, из уравнения 4.30 не так сложно вывести другие свойства умножения элементов базиса кватернионов (попробуйте сделать это самостоятельно):

$$i \cdot j = k = -k \cdot j,$$

$$j \cdot k = i = -i \cdot k,$$

$$k \cdot i = j = -j \cdot i.$$

Теперь я хочу оговорить несколько соглашений, которые будут использоваться при записи кватернионов. Зачастую кватернионы записывают с использованием строчных символов, разбивая кватернионы на действительную и мнимую части, причем мнимая часть записывается в соответствии с соглашениями, принятыми для записи векторов;

$$q = q_0 + q_v, \text{ где } q_v = q_1 \cdot i + q_2 \cdot j + q_3 \cdot k.$$

Итак,  $q_0$  — это действительная часть кватерниона, а  $q_v$  — мнимая часть. Но такое представление, на мой взгляд, несколько противоречит работе с кватернионами в программах, где для их представления используется массив, в котором первый элемент является действительной частью кватерниона, а остальные три — мнимыми коэффициентами. В этом смысле логичнее записывать кватернион как обычную сумму, не выделяя элементы базиса полужирным шрифтом:  $a = -1 + 3 \cdot i + 4 \cdot j + 5 \cdot k$ , либо в чисто векторном виде:  $a = \langle -1, 3, 4, 5 \rangle$ . В зависимости от того, что именно мы делаем с кватернионами, в книге могут использоваться оба варианта записи, но я думаю, что это не вызовет у вас никаких трудностей.

Большим достоинством кватернионов (как и любых гиперкомплексных числовых систем) является то, что сложение, умножение, обращение и т.п. математические операции выполняются так же, как и в обычной теории комплексных чисел, просто с большим количеством элементов. Следовательно, можно считать, что вы уже знаете, как работать с кватернионами, и можно переходить к следующим темам. Конечно, я пошутил, и мы все же бегло ознакомимся с тем, как выполняются те или иные математические действия с кватернионами.

### Сложение и вычитание кватернионов

Сложение и вычитание кватернионов осуществляется путем сложения или вычитания действительной и мнимых частей, как и в случае обычных комплексных чисел.

$$q = q_0 + q_v,$$

$$p = p_0 + p_v,$$

$$q + p = (q_0 + p_0) + (q_v + p_v).$$

Например:

$$q = 3 + 4 \cdot i + 5 \cdot j + 6 \cdot k,$$

$$p = -5 + 2 \cdot i + 2 \cdot j - 3 \cdot k,$$

$$\begin{aligned} q + p &= (3 + (-5)) + (4 + 2) \cdot i + (5 + 2) \cdot j + (6 + (-3)) \cdot k = \\ &= -2 + 6 \cdot i + 7 \cdot j + 3 \cdot k. \end{aligned}$$

В векторной форме это сложение будет выглядеть следующим образом:

$$\langle 3, 4, 5, 6 \rangle + \langle -5, 2, 2, -3 \rangle = \langle -2, 6, 7, 3 \rangle.$$

Пока все просто, но далее нам придется все время помнить, что у нас целые три мнимые компоненты. Чтобы не забывать об этом, мы будем использовать запись с отдельным указанием действительной и мнимой частей.

### Аддитивный обратный элемент и аддитивный нулевой элемент

Аддитивным обратным произвольного кватерниона  $q$  является число, которое, будучи добавленным к данному, даст аддитивный нулевой элемент, который в случае кватернионов представляет собой величину

$$q = 0 + 0 \cdot i + 0 \cdot j + 0 \cdot k = \langle 0, 0, 0, 0 \rangle.$$

Очевидно, что аддитивным обратным кватерниона  $q$  является кватернион  $-q = -q_0 - q_v$ , поскольку

$$q + (-q) = (q_0 - q_0) + (q_v - q_v) = 0 + 0 \cdot i + 0 \cdot j + 0 \cdot k.$$

### Умножение кватернионов

Сложение и вычитание кватернионов выполняется просто и компактно, чего не скажешь об умножении. Оно просто, но далеко не компактно — поскольку требуется вы-

полнить попарное умножение всех слагаемых кватерниона с учетом правил умножения элементов базиса. Давайте попробуем и посмотрим, что у нас получится...

Итак, у нас есть сомножители

$$\mathbf{p} = p_0 + p_1 \cdot \mathbf{i} + p_2 \cdot \mathbf{j} + p_3 \cdot \mathbf{k},$$

$$\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k}.$$

Их произведение равно

$$\begin{aligned} \mathbf{p} \cdot \mathbf{q} &= (p_0 + p_1 \cdot \mathbf{i} + p_2 \cdot \mathbf{j} + p_3 \cdot \mathbf{k}) \cdot (q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k}) = \\ &= p_0 \cdot q_0 + p_0 \cdot q_1 \cdot \mathbf{i} + p_0 \cdot q_2 \cdot \mathbf{j} + p_0 \cdot q_3 \cdot \mathbf{k} + \\ &+ p_1 \cdot \mathbf{i} \cdot q_0 + p_1 \cdot \mathbf{i} \cdot q_1 \cdot \mathbf{i} + p_1 \cdot \mathbf{i} \cdot q_2 \cdot \mathbf{j} + p_1 \cdot \mathbf{i} \cdot q_3 \cdot \mathbf{k} + \\ &+ p_2 \cdot \mathbf{j} \cdot q_0 + p_2 \cdot \mathbf{j} \cdot q_1 \cdot \mathbf{i} + p_2 \cdot \mathbf{j} \cdot q_2 \cdot \mathbf{j} + p_2 \cdot \mathbf{j} \cdot q_3 \cdot \mathbf{k} + \\ &+ p_3 \cdot \mathbf{k} \cdot q_0 + p_3 \cdot \mathbf{k} \cdot q_1 \cdot \mathbf{i} + p_3 \cdot \mathbf{k} \cdot q_2 \cdot \mathbf{j} + p_3 \cdot \mathbf{k} \cdot q_3 \cdot \mathbf{k}. \end{aligned}$$

Если у вас наметанный глаз, возможно, вы заметите определенную структуру в полученном выражении — где-то вам попадется скалярное произведение, где-то — векторное... Можно воспользоваться уравнением 4.30 и упростить полученное выражение, но вы сделаете это и сами, а я сразу покажу вам окончательный вариант представления умножения кватернионов.

#### Уравнение 4.31 . Произведение кватернионов

Для кватернионов

$$\mathbf{p} = p_0 + p_1 \cdot \mathbf{i} + p_2 \cdot \mathbf{j} + p_3 \cdot \mathbf{k} = p_0 + \mathbf{p}_v,$$

$$\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k} = q_0 + \mathbf{q}_v,$$

их произведение равно

$$\mathbf{r} = \mathbf{p} \cdot \mathbf{q} = (p_0 \cdot q_0 - (\mathbf{p}_v \cdot \mathbf{q}_v)) + (p_0 \cdot \mathbf{q}_v + q_0 \cdot \mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v) = r_0 + \mathbf{r}_v.$$

**НА ЗАМЕТКУ**

Здесь оператор  $\times$  обозначает стандартное векторное произведение, вычисляемое так, как если бы мнимая часть кватерниона представляла собой **обычный** трехмерный вектор.

Поскольку скалярное произведение векторов является скаляром, а векторное — вектором, первый член  $(p_0 \cdot q_0 - (\mathbf{p}_v \cdot \mathbf{q}_v))$  представляет собой действительную часть произведения  $r_0$ , а член  $(p_0 \cdot \mathbf{q}_v + q_0 \cdot \mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v)$  — мнимую (векторную) часть произведения  $\mathbf{r}_v$ .

Заметим, что мультипликативным единичным элементом (аналогом “1” в мире обычных чисел) у кватернионов является кватернион  $\mathbf{q}_1 = 1 + 0 \cdot \mathbf{i} + 0 \cdot \mathbf{j} + 0 \cdot \mathbf{k}$ , поскольку, как легко убедиться,  $\mathbf{q}_1 \cdot \mathbf{q} = \mathbf{q} \cdot \mathbf{q}_1 = \mathbf{q}$ .

#### Сопряженный кватернион

Сопряженный кватернион  $\mathbf{q}^*$  вычисляется так же, как и сопряженное комплексное число — изменением знака мнимой части кватерниона  $\mathbf{q}_v$ .

#### Уравнение 4.32. Вычисление комплексно сопряженного кватерниона

Для данного кватерниона  $\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k} = q_0 + \mathbf{q}_v$  комплексно сопряженный кватернион вычисляется путем изменения знака мнимой части:

$$\mathbf{q}^* = q_0 - q_1 \cdot \mathbf{i} - q_2 \cdot \mathbf{j} - q_3 \cdot \mathbf{k} = q_0 - \mathbf{q}_v.$$

Давайте теперь найдем произведение кватерниона на сопряженный с использованием уравнения 4.31.

**Уравнение 4.33. Произведение кватерниона на сопряженный кватернион**

$$\begin{aligned}
q \cdot q^* &= (q_0 + q_1 \cdot i + q_2 \cdot j + q_3 \cdot k) \cdot (q_0 - q_1 \cdot i - q_2 \cdot j - q_3 \cdot k) = \\
&= q_0 \cdot q_0 - (q_1 \cdot (-q_1)) + q_0 \cdot q_1 + q_0 \cdot (-q_1) + (q_1 \cdot (-q_1)) = \\
&= q_0^2 + q_1^2 + q_2^2 + q_3^2 + (q_0 \cdot q_1 - q_0 \cdot q_1) + (0) = \\
&= q_0^2 + q_1^2 + q_2^2 + q_3^2.
\end{aligned}$$

Как видите, произведение кватерниона на комплексно сопряженный кватернион дает действительное число, равное сумме квадратов компонентов кватерниона. Это свойство пригодится нам при вычислении нормы кватерниона и поиске мультипликативного обратного кватерниона.

**Норма кватерниона**

Норма кватерниона вычисляется точно так же, как и норма комплексного числа.

**Уравнение 4.34. Норма кватерниона**

$$\begin{aligned}
q &= q_0 + q_1 \cdot i + q_2 \cdot j + q_3 \cdot k, \\
|q| &= \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{q \cdot q^*}.
\end{aligned}$$

$\left(\sum_{\alpha}^{\infty}\right)$

Заметим, что  $(q^* \cdot q) = (q \cdot q^*)$ . Произведение кватерниона на сопряженный кватернион не зависит от порядка сомножителей, что в общем случае произведения кватернионов неверно:  $q \cdot p \neq p \cdot q$ . Кроме того, очевидно выполнение тождества  $q + q^* = 2 \cdot q_0$ .

**Мультипликативный обратный элемент**

Мультипликативный обратный кватернион имеет особое значение для нас, поскольку позже он будет использоваться для упрощения поворота кватернионов. Итак, мультипликативный обратный кватернион  $q^{-1}$  представляет собой кватернион, обладающий следующим свойством:

$$q \cdot q^{-1} = q^{-1} \cdot q = 1.$$

Давайте умножим каждую часть уравнения на комплексно сопряженный кватернион  $q^*$ :

$$(q \cdot q^{-1}) \cdot q^* = (q^{-1} \cdot q) \cdot q^* = q^*.$$

А теперь давайте рассмотрим правую часть уравнения отдельно:

$$(q^{-1} \cdot q) \cdot q^* = q^*$$

или

$$q^{-1} \cdot (q \cdot q^*) = q^*.$$

Но мы уже знаем, что  $q \cdot q^* = |q|^2$ , так что теперь мы легко можем найти мультипликативный обратный кватернион.

**Уравнение 4.35 а. Обращение кватерниона**

$$q^{-1} = q^* / |q|^2.$$

Кроме того, если  $q$  — единичный кватернион, то  $|q|^2 = 1$ , и уравнение 4.35,а можно упростить.

$$q^{-1} = q^*$$

Правда, красивый результат? Это уравнение — главная причина, по которой везде, где можно, для **осуществления** поворота используются кватернионы. В дальнейшем в большинстве случаев мы будем полагать, что все кватернионы — единичные, так что вполне сможем использовать уравнение 4.35б.

## Применение кватернионов

Теперь, после всего, что мы узнали о кватернионах, они наверняка кажутся вам интересными математическими объектами, которые негде применить в реальной жизни. Однако на самом деле это не так, и кватернионы нашли широкое применение в ряде функций трехмерной графики — поворота и интерполяции от одного поворота к другому (рис. 4.44). На рисунке вы видите два направления камеры, определяемые двумя множествами углов относительно осей  $x$ ,  $y$  и  $z$ :  $(\alpha_1, \phi_1, \theta_1)$  и  $(\alpha_2, \phi_2, \theta_2)$ . Неважно, как **именно** мы направляем камеру, но что если мы хотим плавно перейти от одной ориентации камеры к другой? Это можно сделать, например, воспользовавшись линейной интерполяцией углов, но при этом возникнет масса проблем, в частности, неравномерность движения и потеря степени свободы при выравнивании камеры в направлении оси.

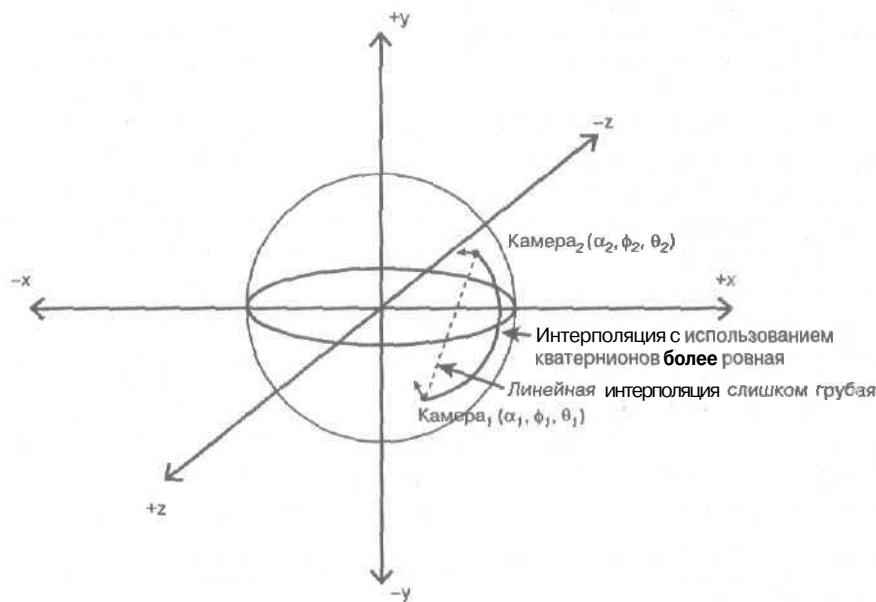


Рис. 4.44. Переход от одной ориентации к другой

Суть заключается в том, что кватернионы, в силу своей четырехмерной природы, могут справиться с этой задачей гораздо более элегантно, чем стандартные углы и матрицы поворота. Мы не будем использовать кватернионы для выполнения стандартных трехмерных поворотов, а оставим их только для серьезной работы с камерой. О деталях такой работы мы поговорим в свое время, а пока просто рассмотрим, как выполняется поворот вектора при помощи кватернионов.

## Поворот кватерниона

Вы можете спросить — какие операторы **могут** быть представлены преобразованиями кватернионов? Попросту **говоря**, можно ли делать с кватернионами что-нибудь интересное? Да, можно. Одна из таких интересных, а главное, полезных операций — **возможность** поворота вектора  $v$ . Мне бы очень хотелось рассказать об этом подробнее, но ни время, ни объем книги не позволяют мне этого, так что мне придется опустить детали в надежде, что фанаты математики найдут соответствующую литературу **сами**, и познакомить вас сразу с конечным результатом. Если нам дан трехмерный вектор  $v = \langle x, y, z \rangle$  (который мы можем записать как кватернион  $v_q = \langle 0, x, y, z \rangle$  при помощи фиктивного компонента  $q_0 = 0$ ), и единичный кватернион  $q$ , то **следующие** операции приведут к повороту  $v_q$ .

### Уравнение 4.36. Поворот кватерниона

Правая система координат

а.  $v'_q = q^* \cdot v_q \cdot q$  — поворот по часовой стрелке,

б.  $v'_q = q \cdot v_q \cdot q^*$  — поворот против часовой стрелки.

Левая система координат

в.  $v'_q = q \cdot v_q \cdot q^*$  — поворот по часовой стрелке.

г.  $v'_q = q^* \cdot v_q \cdot q$  — поворот против часовой стрелки.



Здесь мы используем комплексно сопряженный кватернион в силу единичности кватерниона  $q$ . В противном случае мы должны использовать полное обратное значение  $q^{-1}$ .

Итак,  $v_q$  — вектор, представленный в виде кватерниона с  $q_0 = 0$ , а  $q$  — кватернион. Однако что же на самом деле представляет этот кватернион? Как он связан с осями  $x$ ,  $y$  и  $z$  и с вектором  $v_q$ ? Вас ждет сюрприз!  $q$  представляет собой все — и **ось**, вокруг которой выполняется поворот, и угол поворота вокруг этой оси. Конечно, результат выполнения описанных ранее действий  $v_q$  технически является четырехмерным вектором, или кватернионом. Однако его первый компонент всегда **равен 0**, так что его можно смело отбросить и рассматривать три оставшихся компонента как трехмерный вектор, представляющий исходный вектор после выполнения поворота.

Если попытаться повернуть трехмерный объект вокруг произвольной оси, станет **понятно**, насколько это сложная работа. Но кватернионы существенно облегчают дело. Взгляните на рис. 4.45, где показано, что для данного единичного кватерниона  $q = q_0 + q_v$  ось вращения — просто **прямая**, определяемая векторной частью кватерниона  $q_v$ , а угол поворота  $\theta$  определяется скалярной частью  $q_0$  при помощи следующего соотношения.

### Уравнение 4.37. Преобразование оси и угла в кватернион

$$q = \cos(\theta/2) + \sin(\theta/2) \cdot v_q,$$

$$q_0 = \cos(\theta/2),$$

$$q_v = \sin(\theta/2) \cdot v_q.$$

НА ЗАМЕТКУ

Ясно, что  $v_q$  должен быть единичным вектором, чтобы  $q$  оставался единичным кватернионом.

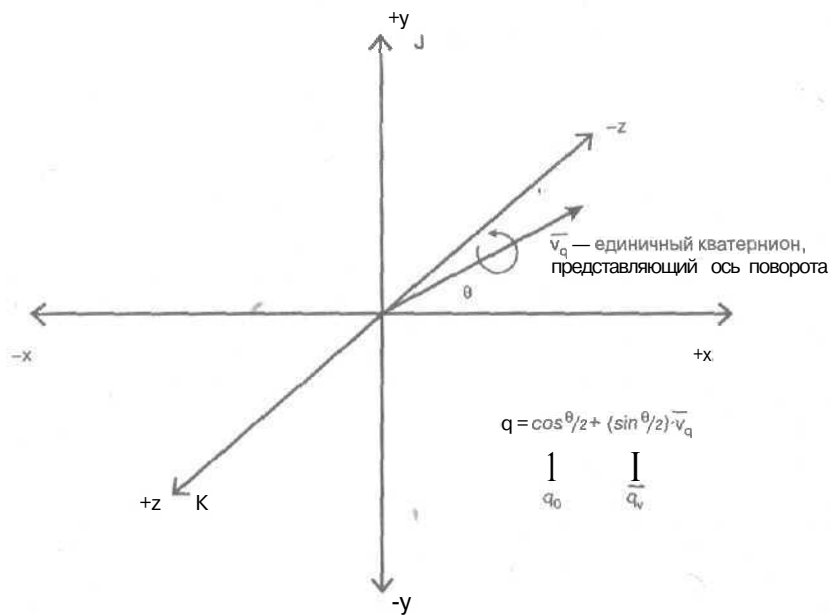


Рис. 4.45. Геометрическая интерпретация кватерниона вращения

А что делать, если у нас нет вектора, вокруг которого надо **осуществить** поворот, Но есть стандартные углы Эйлера? В этом случае можно создать кватернион из углов при помощи одного из произведений:

$$\begin{aligned}
 q_{final} &= q_{x\theta} \cdot q_{y\theta} \cdot q_{z\theta} = \\
 &= q_{x\theta} \cdot q_{z\theta} \cdot q_{y\theta} = \\
 &= q_{y\theta} \cdot q_{x\theta} \cdot q_{z\theta} = \\
 &= q_{y\theta} \cdot q_{z\theta} \cdot q_{x\theta} = \\
 &= q_{z\theta} \cdot q_{x\theta} \cdot q_{y\theta} = \\
 &= q_{z\theta} \cdot q_{y\theta} \cdot q_{x\theta}
 \end{aligned}$$

Здесь  $x_\theta$  обозначает *тангаж* (угол, параллельный оси x),  $y_\theta$  — *рыскание* (угол, параллельный оси y), а  $z_\theta$  — *крен* (угол, параллельный оси z), как показано на рис. 4.46. Все приведенные выше произведения корректны, но наиболее часто используется последнее. В любом случае вы можете подставить необходимые углы в формулы для кватернионов  $q_{x\theta}, q_{y\theta}, q_{z\theta}$  и, выполнив умножение, получить окончательное значение  $q_{final}$ .

#### Уравнение 4.38. Преобразование углов Эйлера в кватернионы

$$\begin{aligned}
 q_{x\theta} &= \cos(x_\theta/2) + \sin(x_\theta/2) \cdot i + 0 \cdot j + 0 \cdot k \\
 q_0 &= \cos(x_\theta/2), \quad q_v = \langle \sin(x_\theta/2), 0, 0 \rangle \\
 q_{y\theta} &= \cos(y_\theta/2) + 0 \cdot i + \sin(y_\theta/2) \cdot j + 0 \cdot k \\
 q_0 &= \cos(y_\theta/2), \quad q_v = \langle 0, \sin(y_\theta/2), 0 \rangle \\
 q_{z\theta} &= \cos(z_\theta/2) + 0 \cdot i + 0 \cdot j + \sin(z_\theta/2) \cdot k \\
 q_0 &= \cos(z_\theta/2), \quad q_v = \langle 0, 0, \sin(z_\theta/2) \rangle
 \end{aligned}$$

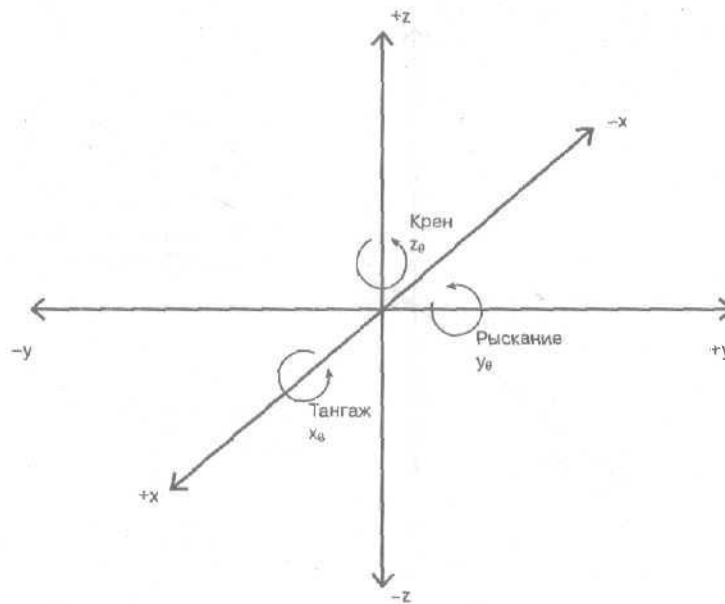


Рис. 4.46. Рыскание, тангаж и крен

Таким образом, для вращения вектора  $v$  с использованием углов Эйлера вам надо выполнить следующие действия:

$$v'_q(0, x', y', z') = (q_{z0} \cdot q_{y0} \cdot q_{x0}) \cdot v_q \cdot (q_{z0} \cdot q_{y0} \cdot q_{x0})^*$$

где  $v_q = \langle 0, x, y, z \rangle$  — начальный вектор или точка, преобразованный для вращения в кватернион.

Ясно, что выражение в скобках  $(q_{z0} \cdot q_{y0} \cdot q_{x0})$  вычисляется только один раз, и в связи со специфичным видом множителей при вычислении происходит много сокращений, так что в конечном счете компоненты этого произведения имеют следующий вид:

$$q_0 = \cos(z_0/2) \cdot \cos(y_0/2) \cdot \cos(x_0/2) + \sin(z_0/2) \cdot \sin(y_0/2) \cdot \sin(x_0/2),$$

$$q_1 = \cos(z_0/2) \cdot \cos(y_0/2) \cdot \sin(x_0/2) - \sin(z_0/2) \cdot \sin(y_0/2) \cdot \cos(x_0/2),$$

$$q_2 = \cos(z_0/2) \cdot \sin(y_0/2) \cdot \cos(x_0/2) + \sin(z_0/2) \cdot \cos(y_0/2) \cdot \sin(x_0/2),$$

$$q_3 = \sin(z_0/2) \cdot \cos(y_0/2) \cdot \cos(x_0/2) - \cos(z_0/2) \cdot \sin(y_0/2) \cdot \sin(x_0/2).$$

**НА ЗАМЕТКУ**

Обратите внимание, что **каждый** кватернион в произведениях можно рассматривать как отдельную операцию поворота. Кроме того, умножение кватернионов менее трудоемко, чем умножение матриц. Таким образом, если у вас выполняется масса умножений матриц, к тому же в цикле, стоит подумать о том, чтобы оптимизировать вашу программу, используя кватернионы, с последующим преобразованием обратно в матрицы.

Преобразование матриц в кватернионы и кватернионов в матрицы — задача сложная, и у меня просто нет возможности решать ее в этой книге. Однако ряд конкретных применений кватернионов в задачах трехмерной графики будет рассмотрен в этой книге позже.

## Дифференциальное исчисление

Я никогда не думал, что в книге, посвященной играм, придется касаться дифференциального исчисления, или, **какого** именовали в свое время, исчисления бесконечно малых. Это исчисление — не более чем математическая теория, независимо разработанная Исааком Ньютоном (Isaac Newton) и Готтфридом Лейбницем (Gottfried Wilhelm Leibnitz) в 17 веке. Дифференциальное исчисление было разработано как инструмент, призванный помочь в решении различных задач движения, описания динамических систем и т.п., где обычная статическая алгебра оказывалась бесполезной. Таким образом, можно сказать, что дифференциальное исчисление — это математика движения.

Поскольку вы намерены заняться программированием игр, сразу скажу, что нечего и думать о создании хорошей современной трехмерной игры, избежав при этом дифференциального исчисления и **дифференциальных** уравнений. Трехмерная графика широко использует линейную алгебру, матрицы и тригонометрию, но игровые процессоры становятся все более и более мощными, а игроки хотят все более красивой графики, обеспечиваемой, например, сплайнами или кривыми **Безье** и т.п. Но даже это не главная проблема — все мы можем прожить и без особого качества кривых в игре, хотя той же математики требует, например, работа с камерой, а математика этих объектов иногда **требует** использования дифференциального исчисления. Но главной проблемой, которая **делает** дифференциальное исчисление в играх абсолютно необходимым, — это *физика*.

Собственно, физика в свое время и вызвала к жизни дифференциальное исчисление, и она же сейчас приводит к его использованию в играх. Сейчас игроки требуют все **большей** степени реалистичности игр, что невозможно без применения массы физических моделей, которые влекут за собой решение сложных математических задач. Избежать использования дифференциального исчисления все равно не удастся, так что лучше, если вы будете подготовлены и в этой области.

Я не так наивен, чтобы надеяться уложить в пару десятков страниц университетский курс высшей математики, но я постараюсь хотя бы познакомить вас с **азами**. Когда я **буду** использовать дифференциальное исчисление в книге, вы можете не понимать в точности, что именно я делаю, но, по крайней мере, после ознакомления с оставшейся частью **данной** главы это уже не приведет вас в состояние прострации. Я надеюсь все же, что большинство читателей этой книги если и не имеют высшего образования, то, по крайней мере, являются студентами, но, тем не менее, я бы не хотел оставить ни одного из читателей без **понимания** хотя бы основных концепций используемой в трехмерных играх математики. Но на **большее** в этой книге рассчитывать не приходится, так что я настойчиво рекомендую изучить кроме моей книги еще и учебник по дифференциальному исчислению.

## Концепция бесконечности

Первая тема, которую я хочу рассмотреть, никак не связана с вашей реальной жизнью, но это реальность в математике — *бесконечность*, обозначаемая символом  $\infty$ . Бесконечность — это одна из тех вещей, о которой все слышали, но никому не удалось, образно говоря, "поддержать ее в руках". В дифференциальном исчислении же бесконечность — обычное понятие, которое не вызывает никаких проблем. Более того, имеются различные виды бесконечностей. Бесконечность — это сугубо математический инструмент.

Как бы то ни было, я хочу, чтобы вы привыкли к бесконечности и научились с ней работать. Давайте начнем с математических операций, которые могут привести нас к бесконечности. Взглянем, например, на такую **дробь**:

$$q = \frac{1}{y}$$

Это вполне нормальная вещь в компьютерной программе, но что случится, если устанет очень маленьким?

$$1/1 = 1, 1/0.1 = 10, 1/0.01 = 100, 1/0.001 = 1000, \dots$$

Как видите, чем меньше становится значение  $u$ , тем больше и больше становится величина частного  $1/u$ . Каким бы я не сделал числитель этой дроби, я всегда могу сделать ее знаменатель гораздо меньшим и получить огромные значения частного. Суть в том, что, каким бы ни был числитель, при уменьшении знаменателя частное неограниченно увеличивается. Каким оно станет, когда в один прекрасный момент знаменатель станет равен нулю? Ответ — плюс бесконечность:

$$\frac{x}{0} = +\infty.$$

Этот ответ не зависит от значения  $x$ . Кроме того, выполняется и обратное условие;

$$\frac{x}{\infty} = 0.$$

Это разумно, что подтверждает рассмотрение такой последовательности дробей:

$$1/1 = 1, 1/10 = 0.1, 1/100 = 0.01, 1/1000 = 0.001, \dots$$

С ростом знаменателя частное становится все меньше и меньше, и в конечном счете при знаменателе, равном бесконечности, частное становится равным нулю.

Есть вопрос, который естественно задать при рассмотрении этих дробей: что, если бесконечно увеличиваются и числитель, и знаменатель дроби? Ответ зависит от того, что из них увеличивается быстрее. Рассмотрим, например, дробь

$$y = \frac{500 \cdot x}{x^2}.$$

Что произойдет с этой дробью при неограниченном возрастании  $x$ ? Числитель будет неограниченно увеличиваться, так же, как и знаменатель. Рассмотрим несколько значений дроби для разных значений  $x$  (см. табл. 4.6).

**Таблица 4.6. Некоторые значения частного  $500 \cdot x / x^2$**

$x$	Числитель $500 \cdot x$	Знаменатель $x^2$	Частное
0	0	0	Не определено
1	500	1	500
2	1000	4	250
3	1500	9	166.66
4	2000	16	125
5	2500	25	100
6	3000	36	83.33
7	3500	49	71.42
8	4000	64	62.5
9	4500	81	55.55
10	5000	100	50

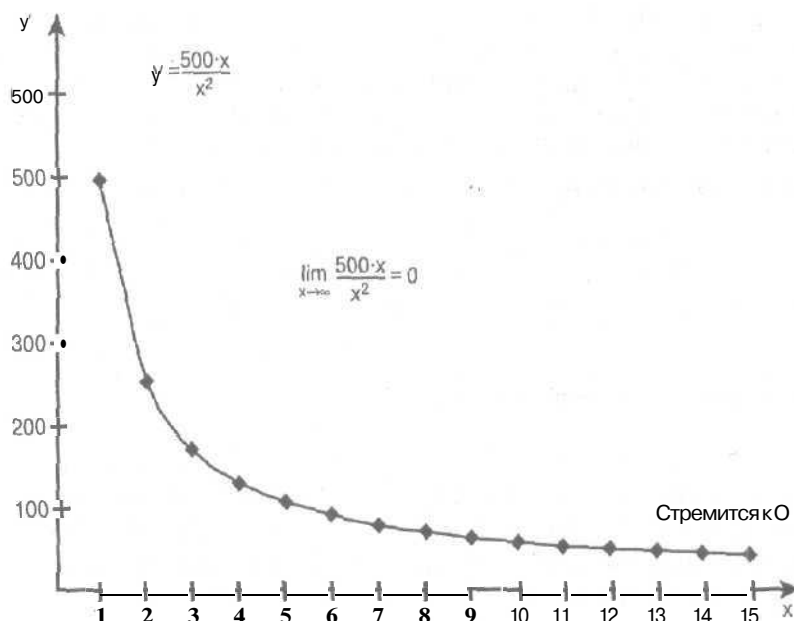


Рис. 4.47. График функции  $500 \cdot x / x^2$

Взгляните также на рис. 4.47. Хотя числитель дроби растет очень быстро, знаменатель растет **еще** быстрее, так что частное становится все меньше и меньше. На самом деле конечный результат, при бесконечном значении  $x$ , будет равен нулю. Заметим, что на первый взгляд, в особенности при **малых** значениях  $x$ , числитель существенно превосходит знаменатель. Однако при больших  $x$  значение  $x^2$  **становится** существенно больше  $500 \cdot x$ , и скорость роста знаменателя превышает скорость роста числителя. **Следовательно**, и в этом случае мы в конечном счете получим нулевое значение. Все эти выводы можно сделать с минимальным использованием алгебры.

Заметим, что если вернуться к исходной дроби и упростить ее путем сокращения  $x$ , то полученный нами результат становится очевиден сразу же:

$$y = \frac{500 \cdot x}{x^2} = \frac{500}{x}.$$

В соответствии с первым рассмотренным примером, мы знаем, что когда  $x$  стремится к бесконечности, то любая конечная величина, деленная на  $x$ , стремится к нулю. Рассмотренный в этом подразделе материал приводит нас к следующей теме — пределам.

## Пределы

Тема пределов очень велика, но в основном она сводится к формализации **того**, о чем мы только что говорили. В общем виде *предел* ~ это математическое выражение, которое помогает нам записать **концепцию** достижения переменной какой-то определенной константы. Это очень близкая к дифференциальному исчислению тема. **Вот** общий вид записи предела:

$$\lim_{x \rightarrow a} f(x) = L.$$

Это можно прочесть как "предел  $f(x)$  при  $x$ , стремящемся к  $a$ , равен  $L$ ". Во многих ситуациях **вычисление** предела выполняется очень просто, но, конечно, нередки к очень

сложные ситуации. Но в данной книге мы не будем углубляться в математические дебри, так что все примеры пределов, с которыми мы будем иметь дело, будут весьма простыми. По сути, для вычисления предела мы будем просто позволять переменной  $x$  (или другой, являющейся переменной предела) стремиться к указанному значению и смотреть на получающийся результат. Все это лучше показать на конкретном примере.

#### Пример 1: простейший предел

$$\lim_{x \rightarrow 3} 2 \cdot x = ?$$

При  $x$ , стремящемся к 3, с  $f(x)$  не происходит ничего необычного, так что мы можем просто подставить вместо  $x$  значение 3, и получить искомый предел;

$$\lim_{x \rightarrow 3} 2 \cdot x = 2 \cdot 3 = 6.$$

#### Пример 2. неопределенный предел

$$\lim_{x \rightarrow 0} \frac{x^2}{3 \cdot x} = ?$$

Этот пример немного сложнее предыдущего. Если подставить значение  $x$ , равное 0, мы получим дробь  $0/0$ , которая не имеет определенного значения. Однако исходную дробь можно упростить, сократив  $x$  в числителе и знаменателе дроби, приводя таким образом исходную задачу к следующей:

$$\lim_{x \rightarrow 0} \frac{x}{3} = ?$$

Решение этой задачи очевидно, поскольку  $0/3 = 0$ .

#### Пример 3: бесконечный предел

$$\lim_{x \rightarrow \infty} \frac{1}{5 \cdot x^2 + 2 \cdot x} = ?$$

С таким примером мы уже сталкивались раньше, когда говорили о бесконечности. В этом случае можно сделать вывод о том, что знаменатель дроби стремится к бесконечности, так что значение дроби стремится к  $1/\infty = 0$ .

#### НА ЗАМЕТКУ

Пределы оказываются особенно полезными, когда вы пытаетесь вывести, чему будет равно некоторое выражение, когда определенная переменная стремится к некоторому значению.

## Суммы и конечные ряды

Следующая тема посвящена не очень сложным вычислениям, но зато она позволит нам более плавно перейти к интегралам. Зачастую при решении итеративных или рекурсивных по своей природе задач мы встречаемся с последовательностью чисел или выражений, подчиняющихся некоторому закону. Например, взгляните на следующую последовательность сложений:

$$1 + 2 + 3 + 4 + \dots + n.$$

Перед тем, как приступить к попыткам вычисления этой суммы, знайте, что имеется более компактный способ ее записи, а именно — с использованием символа суммы  $\Sigma$ . У приведенной выше последовательности есть нижняя граница, равная 1, верхняя, равная  $n$ , и правило, по которому строятся члены этой последовательности. В данном случае правило состоит в прибавлении очередного числа из последовательности, так что мы получаем обычный цикл:

```
for(int count = 1, sum = 0; count <= 1000; ++count)
    sum += count; // Правило вычислений
```

Этот код на языке программирования C/C++ математически записывается следующим образом:

$$\text{sum} = \sum_{n=1}^{1000} n,$$

который в переводе на обычный язык читается как "сумма  $n$  при  $n$ , изменяющемся от 1 до 1000". В общем виде сумма выглядит как

$$\sum_{n=a}^b f(n),$$

где  $a \leq b$  (хотя и необязательно). Значение  $a$  называется нижним пределом суммы, а значение  $b$  — ее верхним пределом.

Вернемся к вопросу вычисления указанной суммы. Пусть  $n$  равно 1 000. Сколько времени вам понадобится, чтобы сложить все эти числа? Это было бы нелегким занятием, если бы не формула для этой суммы (имеются формулы и для множества других сумм), открытая Карлом Гауссом (Karl Gauss),

#### Уравнение 4.39а. Сумма $n$ первых натуральных чисел

$$\sum_{k=1}^n k = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}.$$

Подставляя вместо  $n$  значение 1 000, получим величину 500 500.

Если эта формула произвела на вас впечатление, то чтобы усилить его, сообщу, что Гауссу было всего лишь пять лет, когда он вывел ее. Мы же сейчас приведем еще две формулы, которые могут оказаться вам полезными.

#### Уравнение 4.39б. Сумма $n$ первых квадратов натуральных чисел

$$\sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

#### Уравнение 4.39в. Сумма $n$ первых кубов натуральных чисел

$$\sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \left( \frac{n(n+1)}{2} \right)^2.$$

$$\sum_{k=1}^{\infty} \frac{1}{k^2}$$

Знание этих формул очень полезно при *асимптотическом анализе*, который представляет собой анализ алгоритмов и включает определение времени работы внутренних циклов алгоритма.

Теперь, когда вы знакомы с рядом формул, связанных с суммами, попробуем вычислить суммы, немного отличающиеся от рассмотренных. Например, сумму.

$$\sum_{k=1}^n 3 \cdot k = ?$$

Давайте выпишем несколько членов этой суммы и посмотрим, не натолкнет ли это нас на какие-нибудь идеи.

$$\sum_{k=1}^n 3 \cdot k = 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + \dots + 3 \cdot n.$$

Обратите внимание на множитель 3 у каждого члена суммы. Мы можем вынести его за скобки и получить следующее:

$$\begin{aligned}\sum_{k=1}^n 3 \cdot k &= 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + \dots + 3 \cdot n = \\ &= 3 \cdot (1 + 2 + 3 + \dots + n) = \\ &= 3 \cdot \sum_{k=1}^n k = \frac{3 \cdot n(n+1)}{2}.\end{aligned}$$

Это решение легко обобщить:

$$\sum_{k=1}^n a \cdot k = \frac{a \cdot n(n+1)}{2}.$$

Таким образом мы можем получить множество различных формул для сумм, но это скорее разминка для рук, чем для ума. Так что перейдем к более интересным вещам, а именно — к бесконечным рядам, которые находят применение в физических вычислениях, так что знакомство с ними может помочь вам в вашей карьере программиста трехмерных игр,

## Бесконечные ряды

Конечные ряды, даже самые длинные, когда-нибудь заканчиваются, так что с обычной точки зрения, это вполне реальная вещь, которую, пусть даже за долгое время, но можно вычислить на компьютере. Но в математике встречаются и бесконечные ряды — т.е. ряды, у которых бесконечное число членов, но сумма которых при этом остается конечной. Как такое может быть? Вот тривиальный пример, показывающий, что такое вполне возможно:

$$0 + 0 + 0 + \dots$$

Сумма любого (даже бесконечного) количества нулей равна нулю.

Тем не менее, рассмотренные нами ранее ряды имеют возрастающие члены, сумма которых неограниченно растет. Однако есть и другие ряды, сумма которых стремится к конечному числу. Например, сумма приведенного ниже ряда может стремиться к конечному значению, если величины  $a$  и  $r$  удовлетворяют определенным условиям:

$$\sum_{n=1}^{\infty} a \cdot r^{n-1} = a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \dots + a \cdot r^n + \dots,$$

где  $a$  и  $r$  действительные числа,  $a \neq 0$ . Такой ряд называется *геометрической прогрессией*. Вот пример конкретной геометрической прогрессии:

$$a = 5, \quad r = 1/2$$

$$\begin{aligned}\sum_{n=1}^{\infty} a \cdot r^n &= 5 + 5 \cdot \frac{1}{2} + 5 \cdot \left(\frac{1}{2}\right)^2 + 5 \cdot \left(\frac{1}{2}\right)^3 + \dots = \\ &= 5 + \frac{5}{2} + \frac{5}{4} + \frac{5}{8} + \dots\end{aligned}$$

Здесь четко видно, что члены ряда становятся все меньше и меньше, что подтверждает рис. 4.48. При  $n$ , стремящемся к бесконечности, члены ряда стремятся к нулю, так что становится возможным поверить, что их сумма конечна — и это так и есть!<sup>3</sup>

<sup>3</sup> Только не подумайте, что для того, чтобы ряд имел конечную сумму, достаточно, чтобы его члены стремились к нулю. Например,  $\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \dots = \infty$  — Прим. ред.

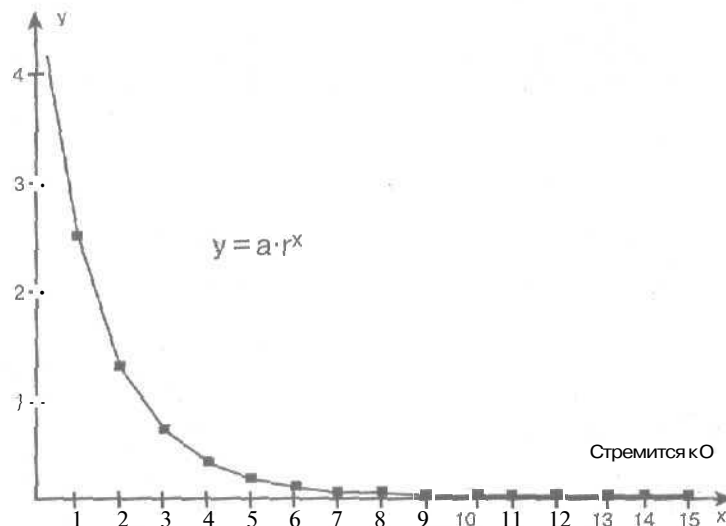


Рис. 4.48. График геометрической прогрессии при  $a=5, r=0.5$

#### Уравнение 4.40. Сумма бесконечной геометрической прогрессии

$$\sum_{n=1}^{\infty} a \cdot r^n = \frac{a}{1-r} \text{ при } |r| < 1.$$

Как видите, для того, чтобы вычислить сумму этого бесконечного ряда, нам вовсе не надо делать бесконечное число вычислений! Нет, все-таки математика — интереснейшая наука... Итак, сумма рассматривавшегося нами ранее ряда с  $a=5, r=0.5$  равна 10.

Ясно, что имеется множество других сходящихся бесконечных рядов, но если вы хотите познакомиться с ними поближе — обратитесь к соответствующим учебникам. Эта книга и без того слишком велика...

## Производные

Наконец, мы вплотную приблизились к дифференциальному исчислению. Однако на самом деле дифференциальное исчисление — это, если можно так выразиться, детство математики — недаром эта тема рассматривается на первом курсе. С другой стороны, я знаю не так много инженеров, кому приходится ежедневно иметь дело с дифференциальным исчислением, и, как ни удивительно, к таким если не сегодня, то завтра наверняка будут относиться разработчики игр. В этом и следующем разделах вы очень (даже чересчур) кратко познакомитесь с производными и интегралами. Сейчас для вас главное — понять идею, а конкретные детали будут дополнительно пояснены там, где эти методы будут применяться на практике. Должен сразу предупредить — не ждите научной строгости в данном изложении материала. Я буду пытаться предельно упростить изложение, но так, чтобы оно оставалось в рамках, позволяющих корректно применять изученный материал.

Вспомним, что идея дифференциального исчисления состоит в том, чтобы описывать и работать с математическими объектами, изменяющимися во времени (или в зависимости от какой-то другой переменной), т.е. с динамическими объектами, которые можно представить в виде функций. Наиболее фундаментальным понятием в описании динами-

ческих объектов является понятие *производной*. Производная функции по сути представляет собой скорость ее изменения. Предположим, например, что у нас в игре есть объект, положение которого описывается формулой  $x = x_0 + \text{vel} \cdot t$  (рис. 4.49).

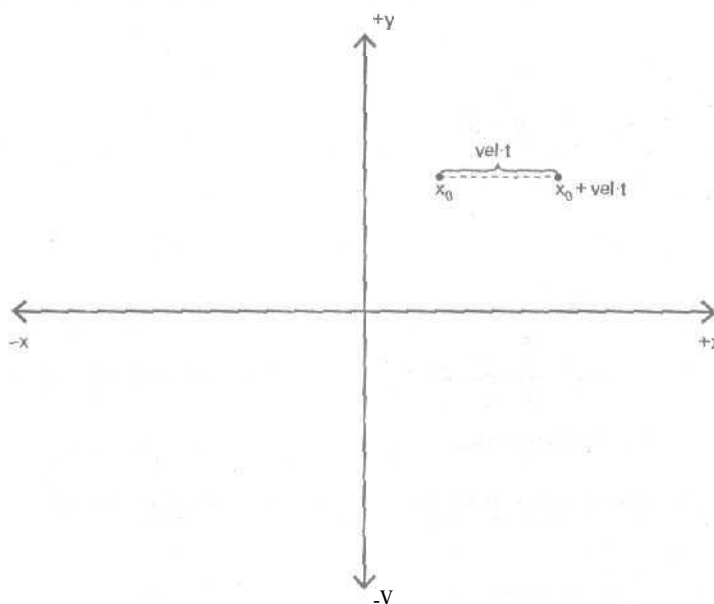


Рис. 4.49. Движущийся по закону  $x = x_0 + \text{vel} \cdot t$  объект

Совершенно ясно, что эта формула описывает координату объекта  $x$  в произвольный момент времени  $t$ , которая равна  $x_0 + \text{vel} \cdot t$ , где  $t$  может быть как временем, так и некоторой искусственной переменной, например, номером кадра. В любом случае, можно задать вопрос — чему равна скорость изменения  $x$ ? Ответ кроется в определении  $x$ , поскольку из него мы знаем, что при каждом увеличении  $t$  на 1, величина  $x$  увеличивается на величину  $\text{vel} \cdot 1$ . Таким образом, скорость изменения  $x$  — величина  $\text{vel}$ . Но что если формула движения окажется немного сложнее? Как математически определить скорость? Ответ заключен в производной.

Если мы продифференцируем  $x$  по  $t$ , то в результате мы получим скорость изменения  $x$  по отношению к  $t$ . Другими словами, производная координаты есть скорость:

$$\text{vel} = \frac{dx}{dt} = f'(t).$$

Не правда ли, очень удобная запись? Осталось выяснить, что же именно она означает.

### Определение производной

Позже я познакомлю вас с рядом других обозначений производной, а пока забудем об обозначениях и сосредоточимся на том, что же такое *производная*. Взгляните на рис. 4.50, на котором изображена кривая, представляющая график функции  $y = f(x)$ . На ней показаны две точки —  $a$  с координатой  $x_0$  и  $b$  с координатой  $x_0 + h$ , где  $h$  — некоторая *небольшая* величина. Через эти две точки проведена *секущая* прямая с некоторым наклоном  $t$ , который определяется по *следующей* формуле.

**Уравнение 4.41. Наклон секущей прямой**

$$m = \frac{\Delta y}{\Delta x} = \frac{f(x_0 + h) - f(x_0)}{h},$$

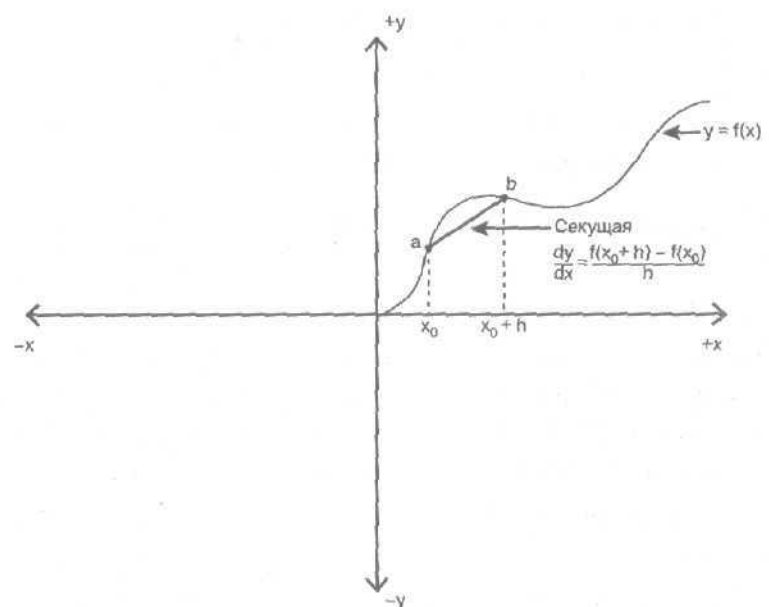


Рис. 4.50. Определение производной

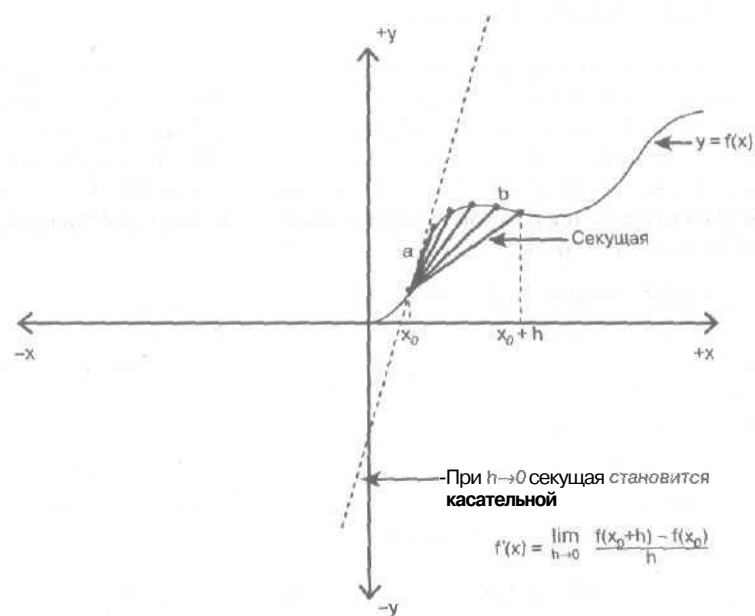


Рис. 4.51. Секущая становится касательной при  $h \rightarrow 0$

Этот наклон представляет собой усредненное значение; нас же интересует точное значение наклона в точке  $x_0$ . Для поиска этого точного значения мы должны уменьшать величину  $h$ , *делая* ее все меньше и меньше, как показано на рис. 4.51. Мы хотим сделать ее бесконечно малой, и тогда *секущая* превратится в *касательную*, наклон которой определяется, как нетрудно *понять*, при *помощи* предела

$$m = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

Когда  $h$  стремится к нулю, предел дает нам значение наклона касательной в точке  $x_0$ , которое и есть производная функции  $f(x)$  в точке  $x_0$ .

#### Уравнение 4.42. Определение производной

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

Да, я уже понял, о чем вы хотите спросить. Запись  $f'(x)$ , или "f штрих", означает "производная  $f$  по  $x$ ", так что штрих — это просто новый способ записи. Аналогично,  $f''(x)$  означает "вторая производная  $f$  по  $x$ ", или производная производной. Перечисленные далее варианты записи обозначают одно и то же.

1.  $f'(x)$  — производная  $f$  по  $x$ . Здесь показана как функция, так и переменная, по которой ее дифференцируют.
2.  $(d/dx) \cdot f(x)$  — та же производная функции  $f$  по  $x$ . В этой записи **акцент** делается на том, что дифференцирование является оператором.
3.  $df/dx$  — аналогично записи 1, но немного по-другому.

Определение производной, пожалуй, самая важная часть дифференцирования. Если указанный предел существует, это именно тот предел, который нас интересует, и который представляет собой значение производной в данной точке. Как же найти этот предел? К счастью, вы пришли не на пустое место, и благодаря трудам множества ученых поиск *производной* стал не таким уж страшным занятием. Имеется масса формул, облегчающих его (поверьте, всякий раз возиться с поиском предела — занятие не из самых веселых). Давайте познакомимся с некоторыми из них.

#### Производные простых полиномов

Вычисление производных полиномов — одна из простейших задач, так что начнем с нее. Первое, что вам надо знать, — это то, что производная суммы функций равна сумме производных:

$$f(x) = f_1(x) + f_2(x) + \dots + f_n(x),$$

$$f'(x) = f_1'(x) + f_2'(x) + \dots + f_n'(x).$$

Для того, чтобы вычислить *производную любого* из членов полинома, надо знать, что

$$(a \cdot x^n)' = a \cdot n \cdot x^{n-1}.$$

Этого достаточно, чтобы найти производную любого полинома. Вот пара конкретных примеров.

### Пример 1

$$f(x) = 5 \cdot x^3; f'(x) = 5 \cdot 3 \cdot x^{3-1} = 15 \cdot x^2.$$

### Пример 2

$$f(x) = 3 \cdot x^2 + 2 \cdot x + 5 = 3 \cdot x^2 + 2 \cdot x^1 + 5 \cdot x^0;$$

$$f'(x) = 3 \cdot 2 \cdot x^{2-1} + 2 \cdot 1 \cdot x^{1-1} + 5 \cdot 0 \cdot x^{0-1} =$$

$$= 6 \cdot x^1 + 2 \cdot x^0 + 0 \cdot x^{-1} = 6 \cdot x + 2.$$

#### НА ЗАМЕТКУ

Обратите внимание, что производная от константы равна нулю.

Далее рассмотрим несколько общих правил вычисления **производных**.

### Производная произведения

Забудьте на минуту, что вы не знаете, как искать производные от конкретных функций. Просто представьте, что у вас есть два выражения, производные от которых вы знаете. Тогда вы легко можете вычислить производную от произведения этих выражений, воспользовавшись следующим правилом:

$$\frac{d}{dx}(u \cdot v) = u \cdot \frac{d}{dx}v + v \cdot \frac{d}{dx}u.$$

В переводе на обычный язык это звучит так: "производная от произведения  $u$  и  $v$  равна  $u$ , умноженному на производную  $v$ , плюс  $v$ , умноженное на производную  $u$ ".

Например, пусть  $f(x) = (3 \cdot x^2 + 3) \cdot (5 \cdot x)$ . Здесь  $u = 3 \cdot x^2 + 3$  и  $v = 5 \cdot x$ . Поэтому

$$f'(x) = (3 \cdot x^2 + 3) \cdot 5 + (5 \cdot x) \cdot (6 \cdot x) = 45 \cdot x^2 + 15.$$

Заметим, что если выполнить умножение в определении  $f(x)$ , то мы можем убедиться в справедливости полученной производной (т.к. мы уже знаем, как дифференцировать полиномы):

$$f(x) = (3 \cdot x^2 + 3) \cdot (5 \cdot x) = 15 \cdot x^3 + 15 \cdot x \text{ и, соответственно,}$$

$$f'(x) = 15 \cdot 3 \cdot x^2 + 15 \cdot x^0 = 45 \cdot x^2 + 15.$$

### Производная частного

Производную частного можно вывести из производной **произведения** и обращения, но чтобы не загружать вас излишне математикой, я просто приведу окончательный результат (впрочем, если хотите — постарайтесь вывести его из правила для **производной произведения** самостоятельно. Это очень просто!):

$$\frac{d}{dx} \frac{u}{v} = \frac{v \frac{d}{dx}u - u \frac{d}{dx}v}{v^2}.$$

Переведите эту математическую запись на обычный язык самостоятельно...

Вот пример использования этой формулы.

$$f(x) = \frac{4 \cdot x^3 - 3}{-3 \cdot x^2}. \text{ Тогда } u = 4 \cdot x^3 - 3, v = -3 \cdot x^2. \text{ Подставляя в формулу для производной ча-}$$

стного значения  $u$  и  $v$ , получим

$$f' = \frac{(-3 \cdot x^2) \cdot (12 \cdot x^2) - (4 \cdot x^3 - 3) \cdot (-6 \cdot x)}{(-3 \cdot x^2)^2} = \frac{-4 \cdot x^4 - 6 \cdot x}{3 \cdot x^4}.$$

## Производная составной функции

Иногда приходится сталкиваться с функциями, которые представляют собой композиции функций. Непонятно? Например, рассмотрим функцию  $f(x) = \sqrt{x^2 + 1} = (x^2 + 1)^{1/2}$ . Как видите, это функция  $x^2 + 1$ , к которой применена функция  $\sqrt{x}$ .



Надеюсь, вы знаете и помните, что квадратный корень — это просто возведение в степень  $1/2$ ?

Искать производную такой функции можно непосредственно из определения производной (не хотите попробовать?), а можно — прибегнув к правилу поиска производной *составной функции*, которая представляет собой композицию функций. В рассматриваемом случае  $g(x) = x^2 + 1$  и  $f(x) = \sqrt{x}$ . Если мы заменим  $x$  в определении функции  $f(x)$  функцией  $g(x)$ , то получим композицию функций  $f$  и  $g$ , или, записывая математически:

$$f(g(x)) = (f \circ g)(x) = \sqrt{x^2 + 1}.$$

НА ЗАМЕТКУ

Обозначения  $f(g(x))$  и  $(f \circ g)(x)$  эквивалентны.

В качестве другого примера возьмем  $f(x) = \sin x$  и  $g(x) = 2 \cdot x + 3$ . Тогда  $(f \circ g)(x) = \sin(2 \cdot x + 3)$ .

Самое приятное при работе с составными функциями заключается в том, что производная такой функции вычисляется достаточно просто; для этого следует вычислить производную внешней функции и умножить на производную внутренней функции. Если внутренняя функция является составной, вычисления продолжаются аналогично. При этом следует заметить, что когда вы находите производную внешней функции, то вместо  $x$  вы подставляете внутреннюю функцию. Математически это правило записывается следующим образом:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dg} f(g) \cdot \frac{d}{dx} g(x).$$

Чтобы все стало просто и понятно, разберем наш многострадальный пример и найдем производную функции  $f(x) = \sqrt{x^2 + 1}$ . Итак,  $f(x) = x^{1/2}$ ,  $g(x) = x^2 + 1$ . Применяя приведенное выше правило поиска производной составной функции, получаем

$$\begin{aligned} f'(g(x)) &= f'(g) \cdot g'(x) = \frac{d}{dg} g^{1/2} \cdot \frac{d}{dx} (x^2 + 1) = \\ &= \frac{1}{2} g^{-1/2} \cdot 2x = x \cdot (x^2 + 1)^{-1/2} = \frac{x}{\sqrt{x^2 + 1}}. \end{aligned}$$

Однако вам придется встречаться не только со степенными функциями и полиномами — в трехмерных играх применение тригонометрических функций не исключение, а правило. Поэтому нам надо обязательно знать производные тригонометрических функций.

## Производные тригонометрических функций

Математический вывод производных тригонометрических функций выходит за рамки нашей книги, поэтому я просто приведу их без вывода и доказательств.

К сожалению, сами по себе, так сказать, в чистом виде, тригонометрические функции встречаются очень редко. Обычно в формулах вы будете сталкиваться с различными коэффициентами при  $x$ , возведением функций в степень, их произведениями и т.п. Однако все это решается — вы знаете, как находить производные произведения, частного, составных функций.

**Таблица 4.7. Производные тригонометрических функций**

Функция	Производная
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\operatorname{tg} x$	$\sec^2 x$
$\operatorname{ctg} x$	$-\operatorname{cosec}^2 x$
$\sec x = 1/\cos x$	$\sec x \cdot \operatorname{tg} x$
$\operatorname{cosec} x = 1/\sin x$	$-\operatorname{cosec} x \cdot \operatorname{ctg} x$

Давайте, например, найдем производную от  $\cos^2 x$ . Представим ее как произведение двух косинусов и применим правило вычисления производной от произведения:

$$f(x) = \cos x \cdot \cos x;$$

$$\begin{aligned} f'(x) &= \cos x \cdot (\cos x)' + (\cos x) \cdot \cos x = \\ &= \cos x \cdot \sin x + \cos x \cdot \sin x = 2 \cdot \cos x \cdot \sin x. \end{aligned}$$

Правда, все очень просто?

Давайте теперь займемся поиском производной функций типа  $f(x) = a \cdot \sin(b \cdot x + c)$ .

Начнем мы с вывода одного общего правила, а именно— с рассмотрения функции  $a \cdot f(x)$ , где  $a$  — константа. Рассматривая это выражение как произведение двух функций и учитывая, что производная от константы равна нулю, получим

$$(a \cdot f(x))' = a \cdot f'(x) + a' \cdot f(x) = a \cdot f'(x) + 0 \cdot f(x) = a \cdot f'(x).$$

Возвращаясь к функции  $f(x) = a \cdot \sin(b \cdot x + c)$ , мы можем сразу сказать, что

$$f'(x) = a \cdot \frac{d}{dx} \sin(b \cdot x + c).$$

Теперь нам надо найти производную функции  $f(x) = \sin(b \cdot x + c)$ . Это очень просто сделать, если заметить, что данную функцию можно рассматривать как составную, где  $f(g) = \sin g$ , а  $g(x) = b \cdot x + c$ . Тогда

$$\begin{aligned} f'(g(x)) &= f'(g) \cdot g'(x) = (\sin g)' \cdot (b \cdot x + c)' = \\ &= \cos(g) \cdot b = b \cdot \cos(b \cdot x + c). \end{aligned}$$

Вспомянув о коэффициенте  $a$ , мы получаем окончательный ответ:

$$(a \cdot \sin(b \cdot x + c))' = a \cdot b \cdot \cos(b \cdot x + c).$$

На этом мы закончим наш маленький экскурс в дифференцирование. После знакомства с материалом данного раздела вы должны уметь дифференцировать полиномы, тригонометрические функции, произведения и частные функций, а также составные функции. Этого в основном вполне достаточно для программирования игр.

## Интегралы

Следующий вопрос, который мы (тоже вкратце) обсудим, — интегрирование. Операция интегрирования обратна к операции дифференцирования. Если мы дифференциру-

ем функцию  $f(x)$  и получаем новую функцию  $f'(x)$ , то как мы можем получить исходную функцию  $f(x)$ , если у нас есть функция  $f'(x)$ ? Этот процесс поиска исходной функции (она называется первообразной) по ее производной и называется *интегрированием*. Имеется ряд *способов* математического определения интегрирования; наряду с ними есть геометрические и физические интерпретации интегрирования, и *потому* я постараюсь познакомить вас с различными точками зрения на интегрирование.

Первый взгляд на интегрирование — через его определение: для данной функции  $f(x)$  найти функцию  $F(x)$  такую, что  $F'(x) = f(x)$ . Рассмотрим, например, функцию  $f(x) = 2 \cdot x$ . Производная от какой функции равна  $2 \cdot x$ ? Поскольку мы только что *занимались* производными, вероятно, вы можете сами дать ответ, что это функция  $F(x) = x^2$ , поскольку

$$\frac{d}{dx} x^2 = 2 \cdot x^{2-1} = 2 \cdot x.$$

Как видите, интегрирование ~ просто обратное действие по сравнению с дифференцированием. По аналогии с только что разобранным примером очевидно, что для  $f(x) = a \cdot x^n$  мы получим функцию  $F(x) = \left( \frac{a}{n+1} \right) \cdot x^{n+1} + c$ . Самостоятельно убедитесь в том, что выполняется условие  $F'(x) = f(x)$ . О том, откуда взялся член  $c$ , я скажу буквально через несколько строк.

Теперь, когда вы на конкретном примере увидели, что такое интегрирование, поговорим о нем как о новом операторе.

### Определение интегрирования

Интегрированием называется процесс вычисления первообразной функции (записывается с использованием знака  $\int$ ):

$$\int f(x) dx = F(x) + c.$$

Здесь  $f(x)$  — функция, для которой мы ищем первообразную,  $F(x)$  — искомая первообразная функция, а  $c$  — константа интегрирования. Давайте посмотрим, откуда взялась эта константа.

Возьмем, например, уже рассмотренную нами функцию  $2 \cdot x$ , первообразной для которой, как мы уже выяснили, является функция  $x^2$ , так как  $(x^2)' = 2 \cdot x$ . Давайте теперь попробуем найти производную функции  $x^2 + 1$ . У вас достаточно знаний, чтобы самостоятельно определить, что  $(x^2 + 1)' = 2 \cdot x$ . Так какая из функций —  $x^2$  или  $x^2 + 1$  — является первообразной для функции  $2 \cdot x$ ? Ведь производные обеих функций одинаковы.

Дело в том, что первообразная функция определяется *с точностью до константы*, т.е. на самом деле существует не одна первообразная функция, производная которой равна исходной, а целое *семейство функций*. Если к любой первообразной функции прибавить константу, получится другая первообразная функция, ничуть не худшая первой. Именно это свойство первообразных и отражено членом " $c$ " в приведенной ранее формуле.

### Геометрическая интерпретация интегрирования

Как я говорил ранее, дифференцирование представляет собой вычисление изменения, или наклона функции, т.е. когда вы дифференцируете функцию  $f(x)$ , вы находите

новую функцию  $f'(x)$ , которая представляет собой не что иное, как значение наклона исходной функции для произвольного  $x$ . Интегрирование имеет подобную геометрическую интерпретацию в обратном смысле. Взгляните на рис. 4.52. На нем вы видите обычную параболическую функцию  $f(x) = x^2$  на отрезке  $[0, 5]$ . Область под кривой заштрихована. Площадь этой области и вычисляется путем интегрирования. Давайте убедимся в этом. Итак, у нас имеется функция  $f(x) = x^2$ . Интегрируя ее (вы уже знаете, как искать интеграл от степенной функции), мы получим

$$\int x^2 dx = \frac{x^3}{3} + c.$$

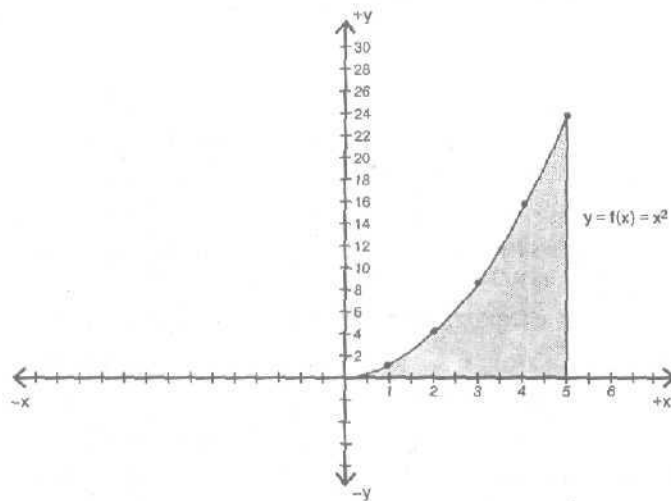


Рис. 4.52. Геометрическая интерпретация интегрирования

Чему же равно  $c$  в данном случае? Здесь нам надо вспомнить о том, что мы рассматриваем исходную функцию на отрезке  $[0, 5]$ . До сих пор мы рассматривали неопределенные интегралы, которые давали нам семейство первообразных функций с неизвестной константой  $c$ , но определенные интегралы позволяют вычислить ее точное значение. Определенный интеграл на отрезке  $[a, b]$  записывается следующим образом.

#### Уравнение 4.43. Определенный интеграл

$$\int_a^b f(x) dx = F(b) - F(a).$$

Отрезок  $[a, b]$  представляет собой пределы интегрирования. Вычисление определенного интеграла производится следующим образом.

Шаг 1. Находим первообразную функцию, принимая значение  $c$  равным какой-то конкретной величине (здесь для простоты мы используем  $c = 0$ ):

$$F(x) = \int x^2 dx = \frac{x^3}{3} + c = \frac{x^3}{3}.$$

Шаг 2. Подставляем в первообразную значения пределов интегрирования и вычитаем первообразную от нижнего предела интегрирования из первообразной от верхнего предела:

$$\int_0^5 x^2 dx = F(5) - F(0) = \frac{5^3}{3} - \frac{0^3}{3} = \frac{125}{3} \approx 41.66.$$

Это и есть значение определенного интеграла, которое в данном случае дает нам площадь под графиком функции  $f(x) = x^2$ . У нас нет другого способа точно вычислить указанную площадь, так что мы оценим ее приближенно, как показано на рис. 4.53 — с помощью аппроксимации треугольником. Площадь треугольника равна половине произведения его основания на высоту, что в данном случае составляет  $0.5 \cdot 5 \cdot 25 = 62.5$  и, как видите, весьма близко к точному значению, вычисленному путем интегрирования.

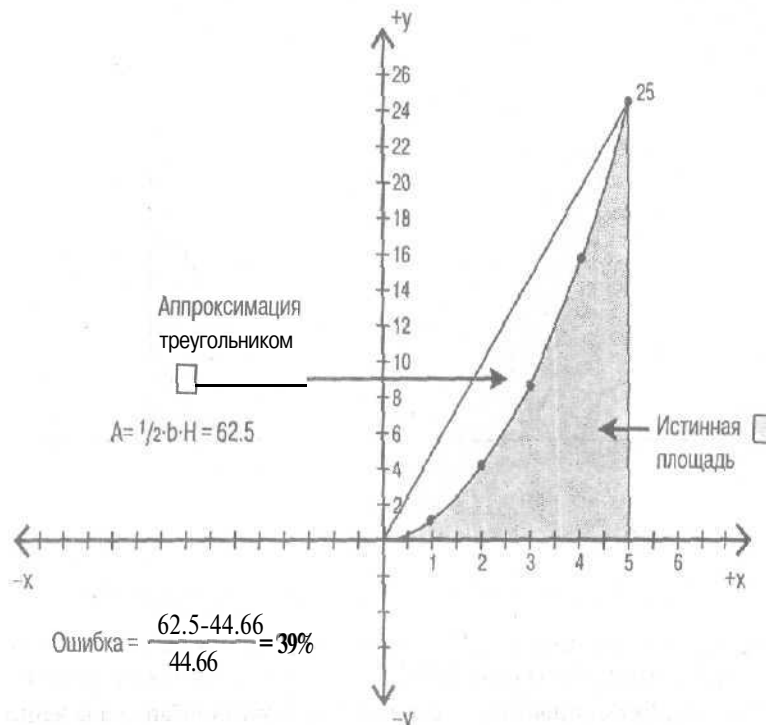


Рис. 4.53. Аппроксимация площади под кривой треугольником

(От редактора. Пожалуй, с методической точки зрения, более поучительно было бы рассмотреть треугольник, образованный прямой  $y = a \cdot x$ , осью  $x$  и прямой  $y = b$ . Понятно, что площадь такого прямоугольного треугольника равна, с одной стороны, половине произведения длин катетов, т.е.  $S = (1/2) \cdot b \cdot a \cdot b = a \cdot b^2/2$ , а с другой — определённому интегралу от функции  $a \cdot x$  на отрезке  $[0, b]$ . Первообразная функции  $a \cdot x$  —  $a \cdot x^2/2$ , так что площадь, определяемая величиной определённого интеграла, равна

$$S = \int_a^b a \cdot x \, dx = F(b) - F(0) = \frac{a \cdot b^2}{2} - \frac{a \cdot 0^2}{2} = \frac{a \cdot b^2}{2}.$$

Как видите, площади, вычисленные при помощи интеграла и геометрическим путем, совпадают, что и является иллюстрацией геометрической интерпретации интегрирования.)

## Формулы интегрирования

Теперь у нас есть математическое определение интегрирования, его геометрическая интерпретация, и мы знаем, как находить неопределенные и определенные интегралы. Чтобы помочь вам в повседневной работе, в табл. 4.8 представлены первообразные некоторых распространенных функций.

**Таблица 4.8. Формулы интегрирования**

Интеграл	Первообразная
$\int a \cdot u^n du$	$a \cdot u^{n+1} / (n+1) + c$
$\int \frac{du}{a \cdot u + b}$	$\ln(a \cdot u + b) / a$
$\int \sin u du$	$-\cos u + c$
$\int \cos u du$	$\sin u + c$
$\int \sec^2 u du$	$\operatorname{tg} u + c$
$\int \operatorname{cosec}^2 u du$	$-\operatorname{ctg} u + c$
$\int \sec u \cdot \operatorname{tg} u du$	$\sec u + c$
$\int \operatorname{cosec} u \cdot \operatorname{ctg} u du$	$-\operatorname{cosec} u + c$
$\int e^{a \cdot u} du$	$e^{a \cdot u} / a + c$
$\int \ln u du$	$u \ln u - u + c$

## Суммирование и физическая интерпретация интегрирования

На самом деле, интегрирование — это попросту суммирование значений функции от одной точки до другой. Это очень полезная операция, абсолютно необходимая при разработке физических моделей. Почему? Потому что очень часто они выглядят аналогично следующей модели:

$$x(t) = x_0 + v \cdot t,$$

$$v(t) = v_0 + a \cdot t,$$

Здесь ускорение  $a$  представляет собой константу. Мы знаем, что производная координаты  $x(t)$  равна скорости, т.е.  $x'(t) = v(t)$ . Это значит, что если у нас есть формула для зависимости скорости от времени, то ее первообразная должна быть равна  $x(t)$ . Интегрируя обе части уравнения, находим:

$$\int x'(t) dt = \int v(t) dt.$$

Левая часть этого уравнения по определению равна  $x(t)$ , поскольку первообразная от производной равна исходной функции. Таким образом, получаем:

$$x(t) = \int v(t) dt.$$

Поскольку мы знаем, что  $v(t) = v_0 + a \cdot t$ , мы можем подставить эту формулу в предыдущую и выполнить интегрирование:

$$x(t) = \int (v_0 + a \cdot t) dt = v_0 \cdot t + a \cdot t^2 / 2 + c.$$

Теперь требуется найти значение  $c$ . Для этого заметим, что в начальный момент времени  $t = 0$  значения координаты и скорости имеют следующие значения:  $x(0) = x_0$  и  $v(t) = v_0$ . Подставляя нулевое значение времени в предыдущую формулу, получаем:

$$0 + 0 + c = x_0.$$

Итак, мы получаем окончательную формулу для равноускоренного движения:

$$x(t) = x_0 + v_0 \cdot t + a \cdot t^2 / 2.$$

Главное в рассмотренном выводе формулы — то, что для нахождения положения объекта мы должны проинтегрировать его скорость. Позже, когда мы перейдем к рассмотрению физических моделей, я покажу вам, каким образом можно выполнять численное интегрирование вместо символического.

## Резюме

Мне никогда не приходилось писать математические книги, но программирование игр заставляет делать это. Я постарался подать материал не "вглубь", а "вширь", чтобы вы могли познакомиться с как можно большим количеством тем — векторами, матрицами, комплексными числами, кватернионами и прочим.

Однако из-за столь большого разнообразия тем преподнести их **сколь-нибудь** глубоко просто невозможно, так что я настоячиво рекомендую вам приобрести учебники по линейной алгебре, дифференциальному исчислению, физике и всерьез изучить эти вопросы. Игры становятся все сложнее и сложнее и по сути близки к переходу в **новую** категорию — имитационного моделирования. Лично я думаю, что лет через пять используемые в играх физические модели будут абсолютно реалистичны — по крайней мере, в пределах классической физики. А это все — математика, математика и еще раз математика! Так что если вы хотите достичь успехов в качестве программиста игр — не ограничивайтесь лишь этой книгой. Без серьезных учебников и **прочных знаний** вам просто не обойтись.

# ГЛАВА 5

## Создание математической библиотеки

### В этой главе...

• Краткий обзор математической библиотеки	310
• Типы и структуры данных	313
• Математические константы	324
• Макросы и встраиваемые функции	327
• Прототипы	336
• API математической библиотеки	341
• Работа математического сопроцессора	390
• Замечания по использованию математической библиотеки	406
» Замечания об оптимизации	407

Первоначально я намеревался заняться созданием математической библиотеки в рамках предыдущей главы, но обычная для этой книги проблема объема материала заставила меня изменить мои планы и посвятить математической библиотеке отдельную главу. В этой библиотеке тем или иным способом должно быть реализовано в виде кода все то, о чем шла речь в главе 4, "Запутанный мир математики". В большинстве случаев рассматривающиеся здесь функции — это просто перевод на язык программирования C/C++ того, что было сказано в предыдущей главе языком математики, Вот схема данной главы:

- краткий обзор математической библиотеки;
- математические константы;
- структуры данных;
- макросы и встраиваемые функции;
- прототипы;
- глобальные переменные;
- системы координат;
- матрицы;
- двумерные и трехмерные прямые;
- трехмерные плоскости;
- кватернионы;
- математика с фиксированной точкой;
- математика *сплывающей* точкой.

## Краткий обзор математической библиотеки

Данная глава является непосредственным продолжением *предыдущей* и в каком-то смысле ее частью — здесь на практике реализуется все то, о чем в предыдущей главе говорилось сугубо теоретически. Разрабатываемая здесь библиотека является частью общей библиотеки, предназначенной для разработки трехмерных игр, которую я буду *постоянно* обновлять, добавлять в нее новые функции и совершенствовать ее код. Отметим, что это не самая эффективная *математическая* библиотека на свете, поскольку я не собираюсь заниматься ее оптимизацией на низком уровне.

Однако тем, где это было возможно, я использовал алгоритмическую оптимизацию высокого уровня — так, например, при вычислении определителя матрицы я использую выделение общих сомножителей вместо вычисления "в лоб", что приводит к несколько меньшему количеству *выполняющихся* умножений. Кроме того, в каждой *игре* интенсивно используется своя математика, так что вы сможете позаботиться о дополнительной оптимизации узких мест при написании собственных игр. Ну, а если вы этого и не делаете — то не сильно беспокоьтесь: наличие процессоров с тактовой частотой свыше 1 ГГц делает эту проблему не такой *ужострой*.

## Структура математической библиотеки

Математическая библиотека относительно проста, так же как и *составляющие* ее файлы. Она состоит всего лишь из двух файлов:

- `T3DLIB4.CPP` — исходный текст на C++;
- `T3DLIB4.H` — заголовочный файл.

На рис. 5.1 представлена схема, *поясняющая взаимоотношения* математической библиотеки с остальными частями нашей системы. Обратите внимание на то, что код в файле математической библиотеки `T3DLIB4.CPP` зависит от некоторых структур из файла `T3DLIB1.CPP`; таким образом, мы должны всегда компоновать эти файлы вместе (впрочем, файл `T3DLIB1.CPP` все равно всегда входит в состав игры, так что данное *требование* по сути ограничением не является). Кроме того, для компиляции файла

T3DLIB1.CPP требуется файл DDRAW.H, так что последний включен с помощью директивы `#include` и в файл T3DLIB4.CPP. Таким образом, если вы хотите использовать рассматриваемую здесь математическую библиотеку отдельно, где-то в другом месте, то вам придется перенести ряд функций и макроопределений из файлов T3DLIB1.CPP и T3DLIB1.H в файлы T3DLIB4.CPP и T3DLIB4.H.

Примечание; должны компоноваться совместно

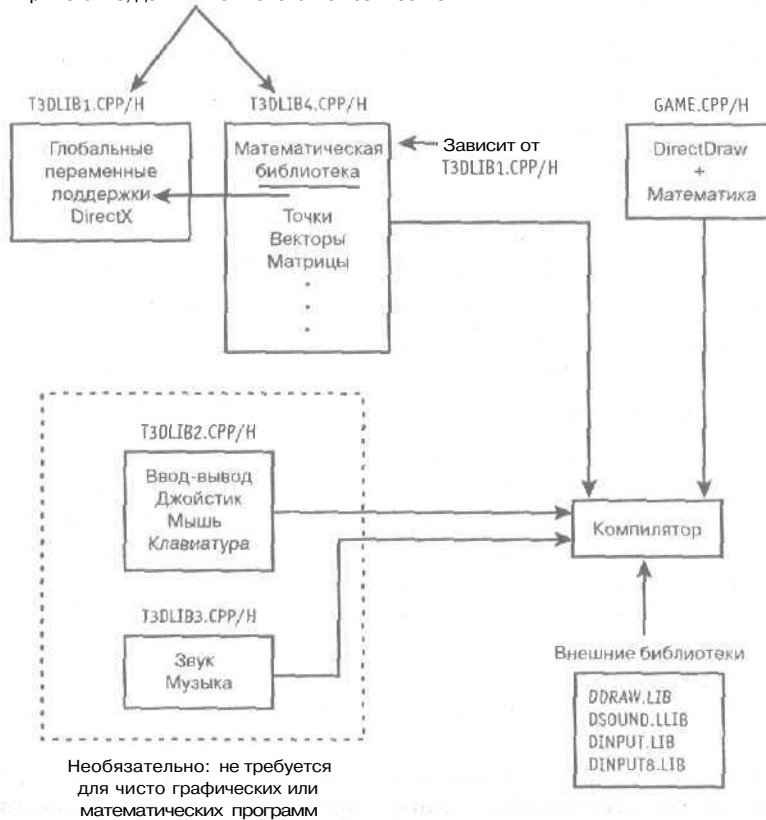


Рис. 5.1. Связь математической библиотеки с другими частями игры

Сама по себе математическая библиотека состоит из большого числа функций (и поверьте, было весьма нелегко протестировать их все), которые помогут вам в работе с точками, векторами, прямыми, матрицами — словом, со всем тем, о чем было рассказано в предыдущей главе. Конечно, удовлетворить все запросы одной библиотекой невозможно, но она дает вам неплохую основу для самостоятельной работы в необходимом для вас направлении.

## Соглашения об именах

Здесь, как и в предыдущей моей книге — *Программирование игр для Windows. Советы профессионала* — при обновлении какой-либо функции я добавляю к ее имени число, указывающее, что хотя функция и выполняет те же действия, что и исходная, но делает это несколько иначе. Например, функция `Multiple()` позже в книге может превратиться

в `Multiple2()`. Именованние функций — задача вообще тяжелая: имя должно быть достаточно понятным, говорящим о том, какая именно функциональность скрыта за ним, но при этом коротким, так как вряд ли вы захотите набирать имена из пары десятков символов. Для того, чтобы имена были разумны, я буду использовать в них, наряду с описанием выполняемых действий имена используемых структур, разделяя отдельные части имени символами подчеркивания (`_`). Например, функция для умножения вектора на матрицу будет выглядеть следующим образом:

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va,
    MATRIX3X3_PTR mb,
    VECTOR3D_PTR vprod);
```

Такое объявление функции вполне понятно говорит о том, для чего она предназначена и с какими типами данных работает. Подобные соглашения об именовании функций, кстати, используются в OpenGL. Согласитесь, что это гораздо приятнее, чем пытаться понять, что же делает функция `MMV3D33(...)`.

НА ЗАМЕТКУ

Кроме того, имена классов и структур у меня всегда используют строчные буквы, так же как и макроопределения `#define`. Однако ряд встраиваемых функций используют те же соглашения об именах, что и обычные функции.

## Обработка ошибок

Обработка ошибок в этой библиотеке предельно проста — потому что ее просто нет! Эта книга предназначена для программистов выше среднего уровня, так что если вам нужна обработка ошибок — позаботьтесь о ней сами. В большинстве случаев функции проверяют наличие самых тупых ошибок, типа деления на ноль, указывая об их наличии возвращаемым значением. Если функция возвращает значение, то обычно 1 (`true`) означает успешное ее выполнение, а 0 (`false`) — наличие проблем.

## Заключительное слово о C++

Годами занимаясь программированием игр и книгами, посвященными этому вопросу, я пришел к окончательному выводу, который вряд ли кому-то удастся поколебать: C++ отличный язык, и я использую его для работы, но для того, чтобы учить других, — он не годится. Вы не задумывались, почему в университетах до сих пор часто учат студентов языкам Pascal, Modula II, Ada? Потому что это хорошо структурированные языки, в которых каждая строка кода имеет единственное точно определенное значение. Разбираться же в программе на C++ без предварительного изучения введенных автором классов — дело весьма сложное. Стоит ли при изучении алгоритма усложнять задачу наличием классов и перегруженных операторов?

Таким образом, поскольку данная книга призвана учить, я буду в основном использовать язык программирования C. Могу только сказать, что вам никто не мешает самостоятельно перевести математическую библиотеку (или любую другую) на C++ для использования в своей повседневной работе.

Что касается лично меня, то я считаю, что работать с векторами или матрицами при помощи перегруженных операторов очень удобно, но совершенно неверно с методической точки зрения. Да и кроме того, при каждом обновлении библиотеки с использованием классов C++ придется вводить новые классы, порождая их из старых или перегружая их. При обновлении функции `Multiple()` я лучше ограничусь тем, что напишу функцию `Multiple2()`.

## Типы и структуры данных

Начнем рассмотрение математической библиотеки с используемых ею типов и структур данных. Большинство типов и структур библиотеки — новые, но в ней используются ряд структур, которые находятся в файлах `T3DLIB1.CPP` и `T3DLIB1.H`, так что в описаниях вам будут время от времени встречаться комментарии, гласящие, что данные типы и структуры можно найти в `T3DLIB1.H`.

Математическая библиотека поддерживает множество типов данных, включая точки, векторы, матрицы, кватернионы и т.п. Мы рассмотрим каждый класс данных отдельно.

### Векторы и точки

Математическая библиотека поддерживает двух-, трех- и четырехмерные точки и векторы, где **четырёхмерность** означает однородные координаты  $(x, y, z, w)$ . Однако в большинстве случаев четвертая координата нами использоваться не будет (она всегда равна 1). Поскольку векторы и точки в действительности обозначают одно и то же, для их хранения используются одни и те же структуры. Кроме того, в структурах я решил использовать объединения, что позволяет использовать различные соглашения об именах. Например, иногда удобно обращаться к координатам трехмерной точки как к `p.x`, `p.y`, `p.z`, а иногда — как к элементам массива: `p.M[0]`, `p.M[1]`, `p.M[2]` или каким-либо иным образом. Этот способ используется мною во всех структурах данных. Вот как это выглядит в реальном коде.

```
// Двумерный вектор или точка, без w ///////////////////////////////////
typedef struct VECTOR2D_TYP
```

```
{
    union
    {
        float M[2]; // Массив для хранения
        // Явно указанные имена
        struct
        {
            float x,y;
        }; // struct
    }; // union
} VECTOR2D, PQINT2D, *VECTOR2D_PTR, *POINT2D_PTR;
```

```
// Трехмерный вектор или точка, без w ///////////////////////////////////
typedef struct VECTOR3D_TYP
```

```
{
    union
    {
        float M[3]; // Массив для хранения
        // Явно указанные имена
        struct
        {
            float x,y,z;
        }; // struct
    }; // union
} VECTOR3D, POINT3D, *VECTOR3D_PTR, *POINT3D_PTR;
```

```
//Четырехмерный однородный вектор или точка (cw)////////
typedef struct VECTOR4D_TYP
(
    union
    {
        float M [4]; // Массив для хранения
        // Явно указанные имена
        struct
        {
            float x,y,z,w;
        }; // struct
    }; // union
} VECTOR4D, POINT4D, *VECTOR4D_PTR, *POINT4D_PTR;
```

Вот несколько старых определений вершин из файла T3DLIB1.H.

```
// Двумерная вершина
typedef struct VERTEX2DI_TYP
{
    int x,y; // Вершина
} VERTEX2DI, *VERTEX2DI_PTR;
```

```
// Двумерная вершина
typedef struct VERTEX2DF_TYP
{
    float x,y; //Вершина
} VERTEX2DF, *VERTEX2DF_PTR;
```

Позже в этой главе, при работе со структурами данных, содержащими и представляющими трехмерные объекты, мы создадим представления для трехмерных вершин.

## Параметризованные прямые

В большинстве случаев алгоритмы используют параметрические представления прямых, создаваемые "на лету", в соответствии с постановкой конкретной задачи. Однако мне кажется, что стоит иметь набор функций для работы с явно заданными параметрическими прямыми. В соответствии с этим я разработал набор функций для двух- и трехмерных параметрических прямых. Вот структура данных, использующаяся при работе с двумерной прямой.

```
// Двумерная параметрическая прямая //////////////////////////////////
typedef struct PARMLINE2D_TYP
{
    POINT2D p0; // Начальная точка
    POINT2D p1; // Конечная точка
    VECTOR2D v; // Вектор направления
    // |v|=|p0->p1|
} PARMLINE2D, *PARMLINE2D_PTR;
```

На рис. 5.2 точка  $p_0$  представляет начальную точку,  $p_1$  — конечную точку, а  $v$  — вектор от точки  $p_0$  к точке  $p_1$ . Из аналогии с двумерной прямой легко понять, что при работе с трехмерной прямой используется следующая структура.

```
// Трехмерная параметрическая прямая //////////////////////////////////
typedef struct PARMLINE3D_TYP
{
```

```

POINT3D p0; // Начальная точка
POINT3D p1; // Конечная точка
VECTOR3D v; // Вектор направления
// |v|=|p0->p1|
} PARMLINE3D, *PARMLINE3D_PTR;

```

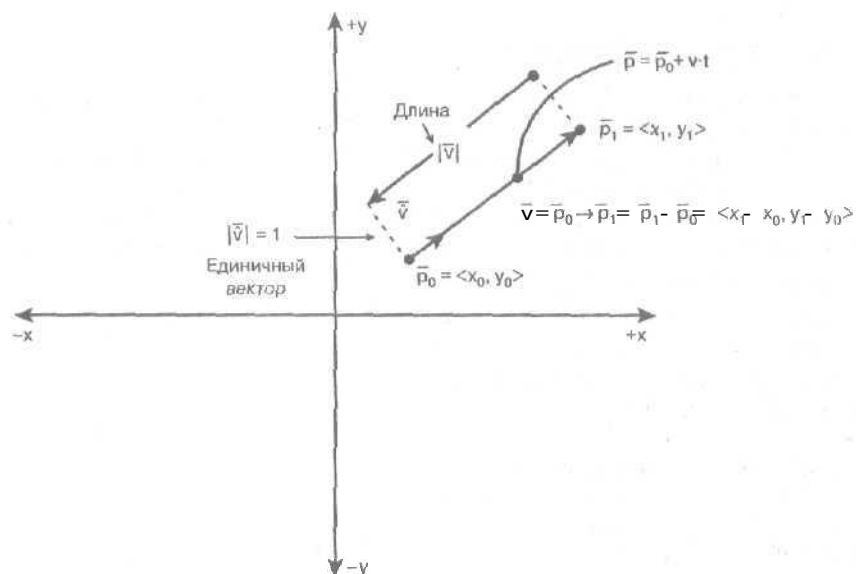


Рис. 5.2. Двумерная параметрическая прямая

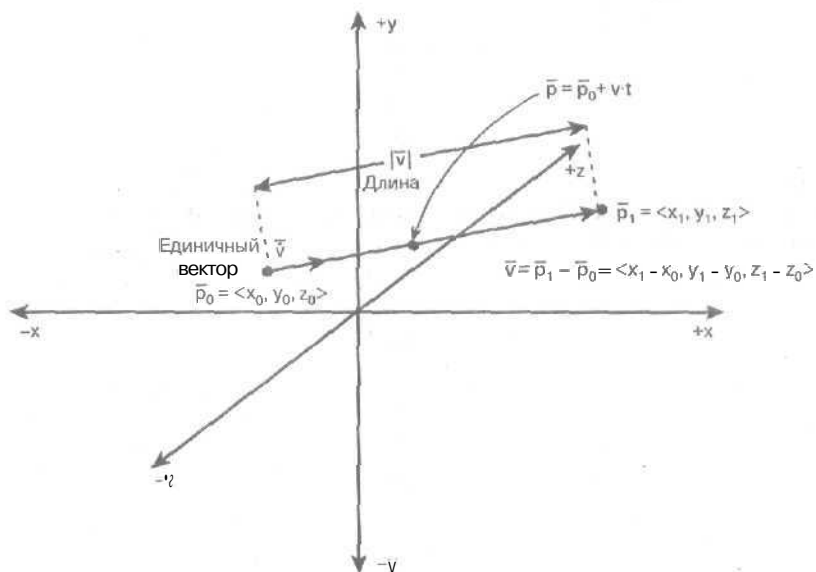


Рис. 5.3. Трёхмерная параметрическая прямая

Трехмерная параметрическая прямая показана на рис. 5.3. Все обозначения на нем те же, что и на рис. 5.2. Обратите внимание, что на рисунке показана левая система координат. Имеет ли это значение? Никакого! Прямая есть прямая, независимо от того, в какой системе координат она рассматривается. Пока вы используете **прямую** в той же системе координат, в которой определяете ее, неприятностей не предвидится.

## Трехмерные плоскости

С трехмерными плоскостями мы сталкиваемся в основном при работе с многоугольниками. Однако я считаю, что в библиотеке надо иметь тип трехмерной плоскости — для работы алгоритмов разбиения, отсечения, определения столкновений и т.п. Имеется несколько способов представления плоскости в трехмерном пространстве, и я выбираю способ определения плоскости при **помощи** вектора нормали и точки. Однако это не означает, что вы не можете использовать в своих разработках другие **виды** представлений, например, в общем виде  $ax + by + cz + d = 0$ .

Итак, определение плоскости при **помощи** точки и нормали естественным образом приводит к следующей структуре данных.

```
//Трехмерная плоскость //////////////////////////////////////
typedef struct PLANE3D_TYP
{
    POINT3D p0; // Точка на плоскости
    VECTOR3D n; // Нормальный (необязательно
                // единичный) вектор
} PLANE3D, *PLANE3D_PTR;
```

На рис. 5.4 изображено рассмотренное представление плоскости в левой системе координат. Здесь  $p_0$  — точка на плоскости, а  $n$  — нормальный вектор. Заметим, что мы не делаем предположений о единичности вектора  $n$ .

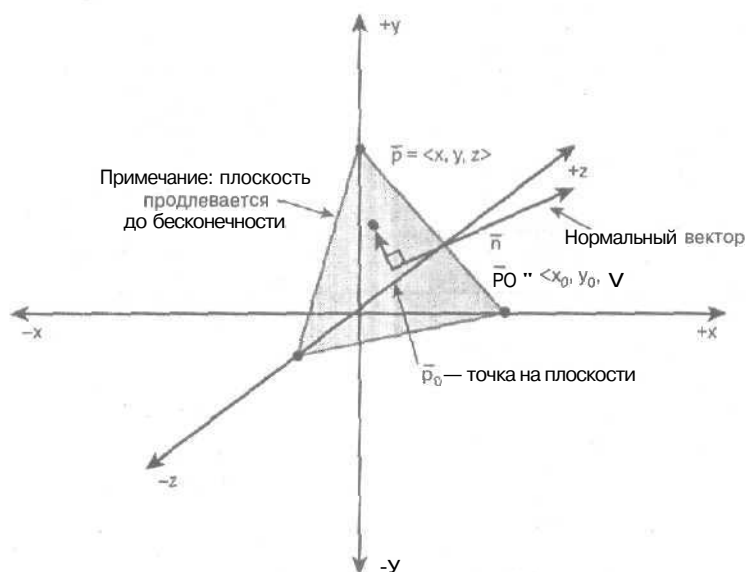


Рис. 5.4. Трехмерная плоскость

## Матрицы

Наибольший набор структур данных в нашей математической библиотеке— это структуры данных для представления **матриц**. Библиотека поддерживает матрицы размера  $1 \times 2$ ,  $2 \times 2$ ,  $1 \times 3$ ,  $3 \times 2$ ,  $3 \times 3$ ,  $1 \times 4$ ,  $4 \times 4$  и  $4 \times 3$ . Матрицы размера  $1 \times n$  представляют собой данные, эквивалентные  $n$ -элементному вектору. Например, матрица  $1 \times 3$ — это те же данные, что и данные, принадлежащие типу VECTOR3D. Это полезно в том плане, что позволяет нам прозрачно преобразовывать один тип данных в другой и использовать один тип данных в функции, написанной для данных другого типа. Замечу, что некоторые матрицы определены в файле T3DLIB1.H, и что все матрицы представлены в виде со старшей строкой и индексируются следующим образом:  $[0 \dots n][0 \dots n] = [\text{строка}][\text{столбец}]$ .

// Определения матриц

// Матрица 4x4 //////////////////////////////////////  
typedef struct MATRIX4X4\_TYP

```
{  
    union  
    {  
        float M[4][4]; // Массив для хранения данных  
        // Явные имена элементов матрицы  
        // в форме со старшей строкой  
        struct  
        {  
            float M00, M01, M02, M03;  
            float M10, M11, M12, M13;  
            float M20, M21, M22, M23;  
            float M30, M31, M32, M33;  
        }; // Конец блока явных имен  
    }; // union  
} MATRIX4X4, *MATRIX4X4_PTR;
```

// Матрица 4x3 //////////////////////////////////////  
typedef struct MATRIX4X3\_TYP

```
{  
    union  
    {  
        float M[4][3]; // Массив для хранения данных  
        // Явные имена элементов матрицы  
        // в форме со старшей строкой  
        struct  
        {  
            float M00, M01, M02;  
            float M10, M11, M12;  
            float M20, M21, M22;  
            float M30, M31, M32;  
        }; // Конец блока явных имен  
    }; // union  
} MATRIX4X3, *MATRIX4X3_PTR;
```

// Матрица 1x4 //////////////////////////////////////  
typedef struct MATRIX1X4\_TYP

```

{
    union
    {
        float M[4]; //Массив для хранения данных
        // Явные имена элементов матрицы
        // в форме со старшей строкой
        struct
        {
            float M00, M01, M02, M03;
        }; // Конец блока явных имен
    }; // union
} MATRIX1X4, *MATRIX1X4_PTR;

// Матрица 3x3 ////////////////////////////////////////
// Из файла T3DLIB1.H
typedef struct MATRIX3X3_TYP
{
    union
    {
        float M[3][3]; //Массив для хранения данных
        // Явные имена элементов матрицы
        // в форме со старшей строкой
        struct
        {
            float M00, M01, M02;
            float M10, M11, M12;
            float M20, M21, M22;
        }; // Конец блока явных имен
    }; // union
} MATRIX3X3, *MATRIX3X3_PTR;

// Матрица 1x3 ////////////////////////////////////////
// Из файла T3DLIB1.H
typedef struct MATRIX1X3_TYP
{
    union
    {
        float M[3]; // Массив для хранения данных
        // Явные имена элементов матрицы
        // в форме со старшей строкой
        struct
        {
            float M00, M01, M02;
        }; // Конец блока явных имен
    }; // union
} MATRIX1X3, *MATRIX1X3_PTR;

// Матрица 3x2 ////////////////////////////////////////
// Из файла T3DLIB1.H
typedef struct MATRIX3X2_TYP
{
    union
    {

```

```

float M[3][2]; // Массив для хранения данных
// Явные имена элементов матрицы
// в форме со старшей строкой
struct
{
    float M00, M01;
    float M10, M11;
    float M20, M21;
}; // Конец блока явных имен
}; // union
} MATRIX3X2, *MATRIX3X2_PTR;

// Матрица 2x2 ////////////////////////////////////////
typedef struct MATRIX2X2_TYP
{
    union
    {
        float M[2][2]; // Массив для хранения данных
        // Явные имена элементов матрицы
        // в форме со старшей строкой
        struct
        {
            float M00, M01;
            float M10, M11;
        }; // Конец блока явных имен
    }; // union
} MATRIX2X2, *MATRIX2X2_PTR;

// Матрица 1x2 ////////////////////////////////////////
// Из файла T3DLIB1.H
typedef struct MATRIX1X2_TYP
{
    union
    {
        float M[2]; // Массив для хранения данных
        // Явные имена элементов матрицы
        // в форме со старшей строкой
        struct
        {
            float M00, M01;
        }; // Конец блока явных имен
    }; // union
} MATRIX1X2, *MATRIX1X2_PTR;

```

Обратите внимание на соглашения об именовании всех матриц. Например, если матрица 3x3 определяется следующим образом

$$\text{MATRIX3X3 } m = \{0, 1, 2, 3, 4, 5, 6, 7, 8\};$$

то в памяти матрица будет выглядеть так:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix},$$

т.е. в форме со старшей строкой. Для обращения к элементам матрицы, в силу определения ее структуры данных как объединения, можно использовать две эквивалентные записи. Так, например, для присвоения нижнему правому элементу матрицы значения 100 можно использовать следующие выражения:

```
m.M22 = 100;
m.M[2][2] = 100;
```

## Кватернионы

Следующий тип данных — четырехмерные гиперкомплексные кватернионы. Напомним, что кватернионы могут быть записаны одним из следующих способов:

$q = q_0 + q_1i + q_2j + q_3k$ , или

$q = q_0 + \langle q_1, q_2, q_3 \rangle$ , или

$q = q_0 + q_v$ .

Как бы ни выглядел кватернион, это лишь кортеж из 4 чисел, которые разделены на действительную часть  $q_0$  и мнимую часть-вектор. Кроме того, поскольку мы намерены использовать кватернионы в контексте вращения и работы с камерой, имеет смысл возможность их представления в виде действительного числа  $w$  и векторной части  $\langle x, y, z \rangle$ . Таким образом мы сможем использовать для работы с кватернионами функции, предназначенные для работы с векторами VECTOR3D. С учетом сказанного вот как выглядит тип данных, предназначенный для хранения кватернионов.

```
// Кватернион ////////////////////////////////////////////
// Обратите внимание на разные способы представления
// кватерниона с помощью объединения
typedef struct QUAT_TYP
{
    union
    {
        float M[4]; // Массив для хранения данных
        struct
        {
            float q0; // Действительная часть
            VECTOR3D qv; // Векторная часть xi+yj+zk
        };
        struct
        {
            float w,x,y,z;
        };
    }; // union
} QUAT, *QUAT_PTR;
```

Обратите внимание, что использование объединения дает возможность обращения к кватерниону тремя разными путями: как к массиву, к паре скаляр-вектор и к явно указанным компонентам  $w$ ,  $x$ ,  $y$  и  $z$ . Используемый способ обращения зависит от алгоритма, который использует кватернионы. Вот примеры кода для изменения действительной части кватерниона:

```
QUAT q = {1, 1, 2, 3};
```

```
q.w = 5; // Явное имя компонента
```

```
q.q0 = 5; // Действительная часть пары скаляр-вектор  
q.M[0] = 5; // Элемент массива
```

**НА ЗАМЕТКУ**

Первоначально я намеревался использовать еще один способ представлений кватернионов — при помощи структуры `VECTOR4D`, однако отказался от нее с тем, чтобы сохранить соответствие используемых типов данных математической записи,

**СОВЕТ**

Позже мы сможем создать новый тип для хранения кватернионов, если обнаружим, что другой порядок данных может ускорить обращение к ним или вычисления.

## Угловые системы координат

Нередко встречаются ситуации, когда применение систем координат, **отличных** от декартовой, оказывается более естественным для рассматриваемой задачи. Например, представим, что мы моделируем орудийную башню (рис. 5.5) — в таком случае лучше других нам подойдет полярная система координат, так как нас в первую очередь интересует направление орудия. Поэтому наша математическая библиотека поддерживает угловые двух- и трехмерные системы координат и преобразования между ними. Поддерживаются двумерная *полярная*, трехмерные *цилиндрическая* и *сферическая* системы координат. Все углы выражаются в радианах, и все преобразования выполняются для правых систем координат, так что будьте внимательны при использовании левых систем координат — вы можете легко получить не тот знак, если забудете о том, какую систему координат используете.

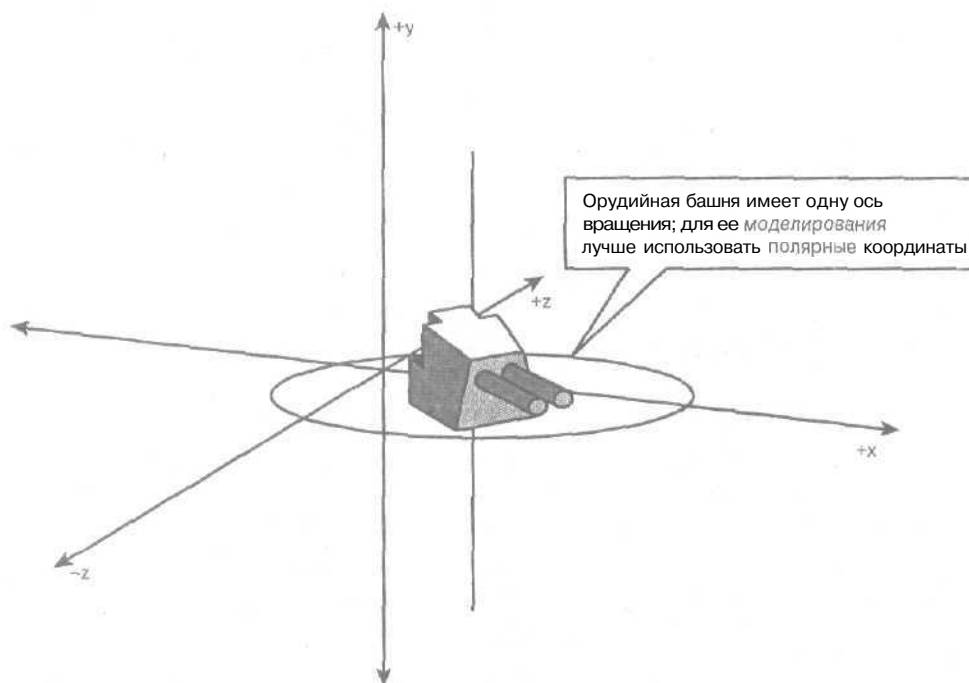


Рис. 5.5. Орудийная башня моделируется в полярной системе координат

**НА ЗАМЕТКУ**

Не забывайте о том, что углы измеряются в **радианах**!

## Двумерные полярные координаты

Двумерная полярная система координат показана на рис. 5.6. Точка в ней представляется при помощи расстояния  $r$  (полярного радиуса) от начала координат (полюса) и полярного угла  $\theta$ . Таким образом, запись  $p(r, \theta)$  означает, что точка расположена по направлению под углом  $\theta$  к полярной оси (в качестве которой обычно принимается ось  $x$ ) в направлении против часовой стрелки, на расстоянии  $r$  от полюса. Соответствующий тип данных представляет собой точный перевод сказанного на язык C++.

```
// Двумерные полярные координаты //////////////////////////////////
typedef struct POLAR2D_TYP
{
    float r; // Полярный радиус
    float theta; // Полярный угол
} POLAR2D; *POLAR2D_PTR;
```

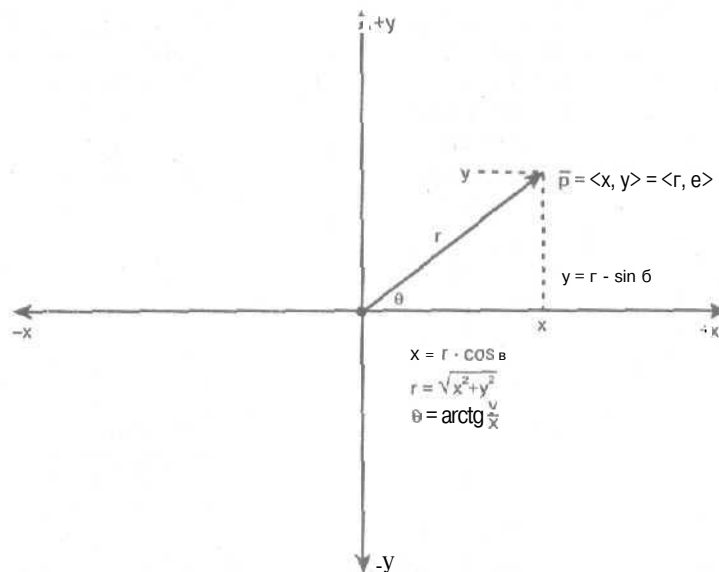


Рис. 5.6. Двумерная полярная система координат

## Трёхмерные цилиндрические координаты

Трёхмерная цилиндрическая система координат похожа на двумерную полярную систему координат, к которой просто добавлена ось  $z$ , как показано на рис. 5.7. Точка в этой системе координат определяется как  $p(r, \theta, z)$ , что означает, что она располагается под углом  $\theta$  относительно полярной оси (в качестве которой обычно принимается ось  $x$ ) в направлении против часовой стрелки, на расстоянии  $r$  от оси  $z$ , и на высоте  $z$  над плоскостью  $xy$  (отрицательные значения  $z$  означают, что точка находится под плоскостью). Соответствующий тип данных представляет собой точный перевод сказанного на язык C++.

```
// Трёхмерные цилиндрические координаты //////////////////////////////////
typedef struct CYLINDRICAL3D_TYP
{
    float r; // Полярный радиус точки
```

```

float theta; // Полярный угол точки
float z; // z-координата точки
} CYLINDRICAL3D, *CYLINDRICAL3D_PTR;

```

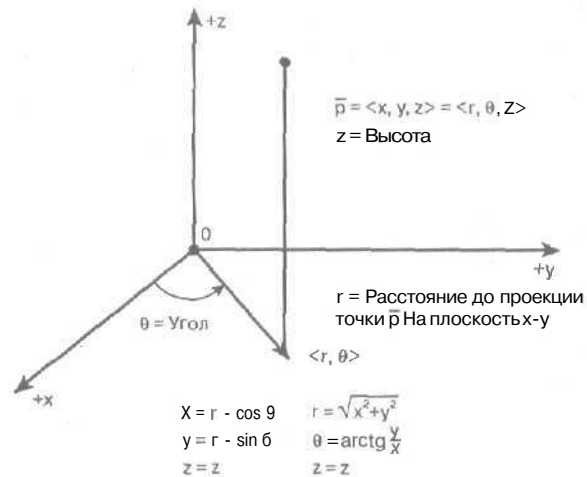


Рис. 5.7. Цилиндрическая система координат

## Трёхмерные сферические координаты

Это наиболее сложная система координат среди всех угловых систем. Точка  $p(p, \theta, \phi)$  определяется расстоянием и двумя углами, как показано на рис. 5.8, где

- $p$  — полярный радиус; расстояние от начала координат (полюса) до точки  $p$ .
- $\theta$  — широта; угол, который направленный отрезок от полюса к точке составляет с положительным направлением оси  $z$ . Этот угол может принимать значения из интервала  $[0, \pi]$ .
- $\phi$  — долгота; угол, который проекция отрезка от полюса к точке на плоскость  $xy$  образует с положительным направлением оси  $x$  (аналог стандартного полярного угла в двумерных полярных координатах).

Соответственно, для хранения сферических координат точки используется следующая структура.

```

// Трёхмерные сферические координаты //////////////////////////////////
typedef struct SPHERICAL3D_TYP
{
    float p; // Полярный радиус
    float theta; // Широта
    float phi; // Долгота
} SPHERICAL3D, *SPHERICAL3D_PTR;

```

## Числа с фиксированной точкой

Числа с фиксированной точкой, как вы знаете, представляют собой целые числа, которые используются для представления дробей или чисел с плавающей точкой с ограниченной точностью, для работы с которыми не требуется сопроцессор. В настоящее время

необходимость в таком представлении очень спорна, поскольку современные процессоры Pentium+ и I64 выполняют операции с числами с плавающей точкой с той же скоростью, что и с целыми числами.

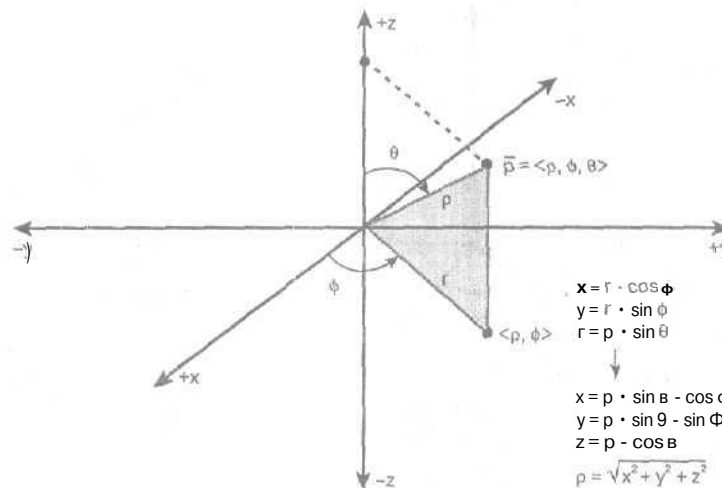


Рис. 5.8. Трехмерные сферические координаты

Тем не менее, в некоторых ситуациях все же имеет смысл использовать числа с фиксированной точкой — например, в глубоко вложенных внутренних циклах, либо если вы хотите использовать исключительно целые числа; возможны и другие ситуации, где их применение целесообразно. В данной книге мы будем использовать числа с фиксированной точкой в формате 16.16, показанном на рис. 5.9— 16 битов при этом представляют целую часть числа, а вторые 16 — дробную. Вот как выглядит соответствующий тип.

```
// Числа с фиксированной точкой //////////////////////////////////
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

Теперь пришло время познакомиться с математическими константами, используемыми в нашей математической библиотеке. Обычно ознакомление с ними выполняется в первую очередь, но я решил, что будет более полезно сначала изучить используемые в библиотеке типы данных.

## Математические константы

Математическая библиотека использует ряд констант, которые определены в файле T3DLIB4.H, и некоторые константы, определенные в файле T3DLIB1.H. Начнем с констант из последнего файла.

```
// Математические константы из файла T3DLIB1.H
```

```
// Число "пи" и связанные с ним величины
#define PI ((float)3.141592654f)
#define PI2 ((float)6.283185307f)
#define PI_DIV_2 ((float)1.570796327f)
#define PI_DIV_4 ((float)0.785398163f)
```

```
#define PI_INV ((float)0.318309886f)

// Константы, связанные с числами с фиксированной точкой
#define FIXP16_SHIFT 16
#define FIXP16_MAG 65536
#define FIXP16_DP_MASK 0x0000ffff
#define FIXP16_WP_MASK 0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

#### а. 16-битовый формат с фиксированной точкой



#### б. 32-битовый формат с фиксированной точкой

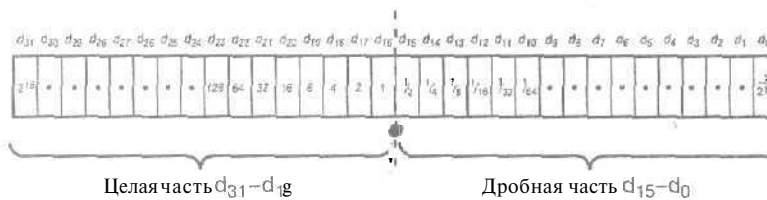


Рис. 5.9. Представления чисел с фиксированной точкой

Как видите, в файле T30LIB1.H приведены только константы, связанные с числом  $\pi$  и с представлением чисел в формате с фиксированной точкой. Напомню, что мы используем числа с фиксированной точкой в формате 16.16.

Теперь мы перейдем к рассмотрению констант из файла T3DLIB4.H.

```
// Определение малых чисел
```

```
#define EPSILON_E4 (float)(1E-4)
#define EPSILON_E5 (float)(1E-5)
#define EPSILON_E6 (float)(1E-6)
```

Эти  $\epsilon$ -константы помогают осуществлять сравнение чисел с плавающей точкой. Например, достаточно распространенной является операция проверки некоторого значения с плавающей точкой на равенство 0. Однако в математике с плавающей точкой непосредственное сравнение со значением 0.0 некорректно в силу ряда причин, в частности, потери точности при вычислениях или в силу ограничений, накладываемых представлением чисел, поэтому такое сравнение выполняется проверкой степени близости числа к нулевому значению.

```
if (fabs(x) < 0.00001)
{
```

```
// x достаточно близко к 0
} // if
```

Следующий набор констант используется в функциях, работающих с параметрическими прямыми, в частности, в качестве возвращаемых некоторыми функциями значений (подробнее об этом рассказывается при рассмотрении конкретных функций).

// Значения, определяющие пересечения отрезков

```
#define PARM_LINE_NO_INTERSECT 0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

Еще один вид констант — определения единичных матриц разного размера.

// Единичные матрицы

// Единичная матрица 4x4  
const MATRIX4X4 IMAT\_4X4 =

```
{ 1,0,0,0,
  0,1,0,0,
  0,0,1,0,
  0,0,0,1
```

```
};
```

// Единичная матрица 4x3 (с математической точки зрения некорректна, но позже будет использоваться в предположении, что четвертый столбец всегда равен  $[0 \ 0 \ 0 \ 1]^T$ )

const MATRIX4X3 IMAT\_4X3 =

```
{ 1,0,0,
  0,1,0,
  0,0,1,
  0,0,0
```

```
};
```

// Единичная матрица 3x3  
const MATRIX3X3 IMAT\_3X3 =

```
{ 1,0,0,
  0,1,0,
  0,0,1
```

```
};
```

// Единичная матрица 2x2  
const MATRIX2X2 IMAT\_2X2 =

```
{ 1,0,
  0,1
```

```
};
```

**НА ЗАМЕТКУ**

Хотя единичными могут быть только квадратные матрицы, т.е. матрицы размера 2x2, 3x3 и т.д., здесь представлена единичная матрица размером 4x3. С математической точки зрения, это нонсенс, но мы будем использовать эту матрицу, считая, что последний ее столбец равен  $[0 \ 0 \ 0 \ 1]^T$ .

## Макросы и встраиваемые функции

Большой проблемой при написании эффективной математической библиотеки становится то, что нередко вызов функции (без учета времени ее выполнения) для выполнения некоторых операций над математическим объектом производится дольше, чем сами вычисления. Следовательно, определенную роль в повышении эффективности библиотеки может сыграть широкое применение макросов и встраиваемых функций. Но дело в том, что встраиваемые функции должны быть доступны во время компиляции, а не во время компоновки. Обычно, когда компилятор в исходном тексте встречается вызов функции `func()` из другого модуля, компилятору для генерации кода вызова функции достаточно знать только ее прототип, но не сам код этой функции. Но если это встраиваемая функция, ситуация изменяется, и компилятор должен иметь доступ к исходному тексту данной функции. Таким образом, единственный способ реализации больших встраиваемых функций — наличие заголовочного файла с текстами функций, который включается в программу при помощи директивы препроцессора `#include`. Схема использования обычных и встраиваемых функций приведена на рис. 5.10.

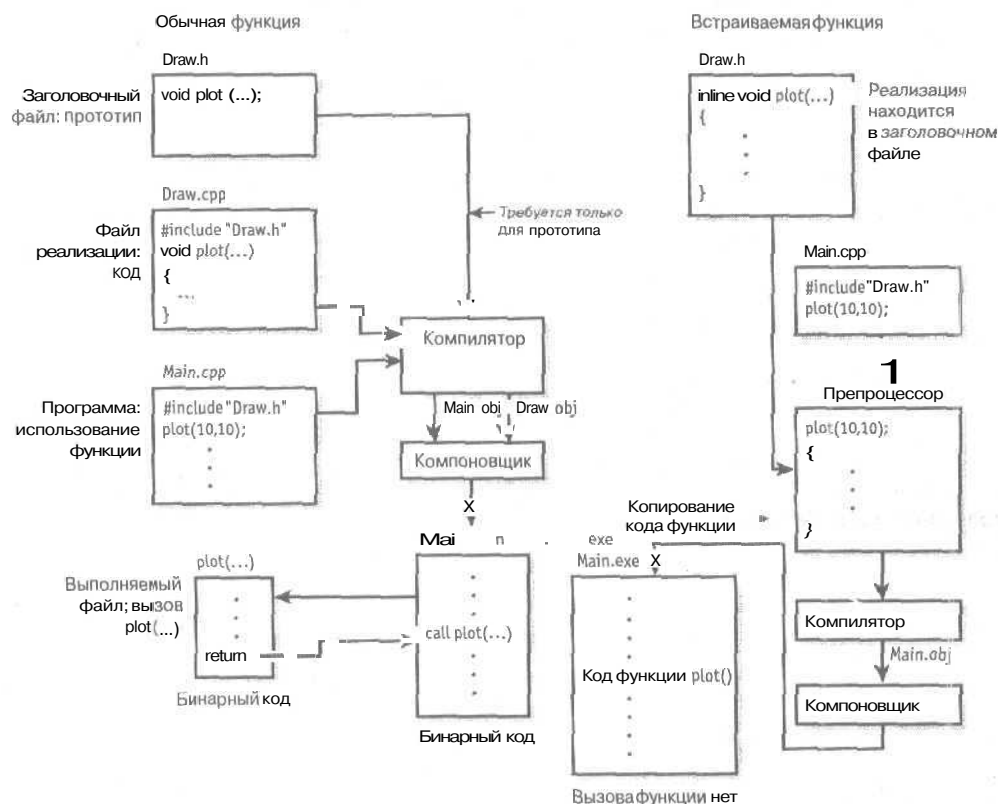


Рис. 5.10. Использование обычных и встраиваемых функций

Итак, главный вывод — при написании математической библиотеки функции в большинстве своем должны быть **встраиваемыми**, что позволит обойтись без осуществления вызова функций. К чему это приводит, рассмотрим на примере вызова следующей функции во внутреннем цикле.

```
void Plot_Pixel(UCHAR *video_buffer, int mempitch,
               int x, int y, int color)
```

```
{
    // Вывод пикселя
    video_buffer[x+y*mempitch] = color;
} // Plot_Pixel
```

Если я **осуществляю** вывод в своей программе, генерируется вызов функции. Например, рассмотрим **следующий** код.

```
void main()
{
    UCHAR *video_buffer =
        (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);
    for (int y=0; y < SCREEN_HEIGHT; y++)
        for (int x=0; x < SCREEN_WIDTH; x++)
            Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);
} // main
```

Хотя все в этой функции выглядит вполне корректно, давайте посмотрим, как выглядит **дизассемблированный** код, сгенерированный Visual C++ 6.0.

```
_main PROC NEAR

;52:{

    push ebp
    mov ebp, esp
    sub esp, 12                ; 0000000cH

; 53 : UCHAR *video_buffer =
;      (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push 307200                ; 0004b000H
    call _malloc
    add esp, 4
    mov DWORD PTR _video_buffer$[ebp], eax

; 54 :
; 55 : for (int y=0; y < SCREEN_HEIGHT; y++)

    mov DWORD PTR _y$[ebp], 0
    jmp SHORT $L43914
$L43915:
    mov eax, DWORD PTR _y$[ebp]
    add eax, 1
    mov DWORD PTR _y$[ebp], eax
$L43914:
    cmp DWORD PTR _y$[ebp], 480 ; 000001e0H
    jge SHORT $L43916

; 56 : for (int x=0; x < SCREEN_WIDTH; x++)

    mov DWORD PTR _x$43917[ebp], 0
    jmp SHORT $L43918
```

```

$L43919:
    mov ecx, DWORD PTR _x$43917[ebp]
    add ecx, 1
    mov DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp DWORD PTR _x$43917[ebp], 640 ; 00000280H
    jge SHORT $L43920

; 57 : Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);

    push 5
    mov edx, DWORD PTR _y$[ebp]
    push edx
    mov eax, DWORD PTR _x$43917[ebp]
    push eax
    push 640 ; 00000280H
    mov ecx, DWORD PTR _video_buffer$[ebp]
    push ecx
    call ?Plot_Pixel@@YAXPAHHHH@Z ; Plot_Pixel
    add esp, 20 ; 00000014H
    jmp SHORT $L43919
$L43920:
    jmp SHORT $L43915
$L43916:

; 58 :
; 59 : } // end main

    mov esp, ebp
    pop ebp
    ret 0
_main ENDP

    Здесь полужирным шрифтом выделен весь код, необходимый для вызова функции
    Plot_Pixel() — настройка кадра стека, вызов, сброс стека после вызова. Здесь не учтен код
    самой функции, который выглядит следующим образом.
    ?Plot_Pixel@@YAXPAHHHH@Z PROC NEAR ; Plot_Pixel

; 45 : {

    push ebp
    mov ebp, esp

; 46 : //plots a pixel
; 47 : video_buffer[x+y*mempitch] = color;

    mov eax, DWORD PTR _y$[ebp]
    imul eax, DWORD PTR _mempitch$[ebp]
    mov ecx, DWORD PTR _x$[ebp]
    add ecx, eax
    mov edx, DWORD PTR _video_buffer$[ebp]
    mov al, BYTE PTR _color$[ebp]

```

```

mov BYTE PTR [edx+ecx], al
; 48 : } // Plot_Pixel

pop ebp
ret 0
?Plot_Pixel@@YAXPAHHHH@Z ENDP ; Plot_Pixel

```

Как видите, вызов Plot\_Pixel() делает массу работы, аналогичной работе при вызове функции! Однако, если сделать эту функцию встраиваемой, все это окажется **излишним**. В этом легко убедиться, объявив функцию встраиваемой и **дизассемблировав** сгенерированный код.

```

_main PROC NEAR

;52:{

    push ebp
    mov ebp, esp
    sub esp, 16 ; 00000010H

; 53 : UCHAR *video_buffer =
    (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push 307200 ; 0004b000H
    call _malloc
    add esp, 4
    mov DWORD PTR _video_buffer$[ebp], eax

; 54 :
; 55 : for(int y=0; y<SCREEN_HEIGHT; y++)

    mov DWORD PTR _y$[ebp], 0
    jmp SHORT $L43914
$L43915:
    mov eax, DWORD PTR _y$[ebp]
    add eax, 1
    mov DWORD PTR _y$[ebp], eax
$L43914:
    cmp DWORD PTR _y$[ebp], 480 ; 000001e0H
    jge SHORT $L43916

; 56 : for(int x=0; x<SCREEN_WIDTH; x++)

    mov DWORD PTR _x$43917[ebp], 0
    jmp SHORT $L43918
$L43919:
    mov ecx, DWORD PTR _x$43917[ebp]
    add ecx, 1
    mov DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp DWORD PTR _x$43917[ebp], 640 ; 00000280H
    jge SHORT $L43920

```

```
; 57 : Plot_Pixel(video_buffer, SCREEN_WIDTH,x,y,5);
```

```
mov DWORD PTR $T43941[ebp], 5
mov edx, DWORD PTR _y$[ebp]
imul edx, 640 ; 00000280H
mov eax, DWORD PTR _x$43917[ebp]
add eax, edx
mov ecx, DWORD PTR _video_buffer$[ebp]
mov dl, BYTE PTR $T43941[ebp]
mov BYTE PTR [ecx+eax], dl
jmp SHORT $L43919
$L43920:
jmp SHORT $L43915
$L43916:
```

```
; 58 :
; 59 : } // main
```

Здесь полужирным шрифтом выделен встроенный вызов функции `Plot_Pixel()` — как видите, ситуация существенно улучшилась. Код стал несколько большим, но зато быстрее примерно в два раза.

Думаю, что я наглядно показал преимущества использования встраиваемых функций — естественно, когда эти функции имеют небольшой размер. Однако, к сожалению, код встраиваемых функций должен быть доступен компилятору при их использовании, так что единственное возможное место их размещения — заголовочные файлы. Однако это не слишком удобно для программиста, поскольку приводит к быстрому росту файла (и, как следствие, к сложности работы с ним и замедлению работы компилятора).

В результате программист вынужден искать золотую середину и решать, какие функции будут встраиваемыми и размещены в `.H`-файле, а какие — обычными и размещены в `.CPP`-файле, опираясь на интуицию и здравый смысл. Я размещаю в `.H`-файле только макроопределения и простейшие встраиваемые функции. Вы можете разместить их по-своему.

## Утилиты общего назначения и преобразование величин

Все эти макросы располагаются в файле `T3DLIB1.H`.

```
// Вычисление минимального и максимального
// значения двух выражений
#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAX(a,b) (((a) > (b)) ? (b) : (a))

// Обмен значениями
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// Некоторые математические макросы
#define DEG_TO_RAD(ang) ((ang)*PI/180.0)
#define RAD_TO_DEG(rads) ((rads)*180.0/PI)

#define RAND_RANGE(x,y) ( (x) + (rand()%((y)-(x)+1)))
```

Работа каждого макроса очевидна, так что пояснять тут ничего не требуется.

## Точки и векторы

Как я упоминал ранее в данной главе, формат данных для представления точек и векторов одинаков, и поэтому группирую макросы для точек и векторов вместе. Кроме того, ряд макросов представляют собой встраиваемые функции, что обеспечивает выполнение проверки типов компилятором. Да и к тому же, многие из них просто слишком велики для записи в качестве макроопределения `#define`.

```
// Макросы для работы с векторами (обратите внимание на
// инициализацию значения поля w четырехмерного вектора)
```

```
// Обнуление векторов
```

```
inline void VECTOR2D_ZERO(VECTOR2D_PTR v)
{(v)->x = (v)->y = 0.0;}
```

```
inline void VECTOR3D_ZERO(VECTOR3D_PTR v)
{(v)->x = (v)->y = (v)->z = 0.0;}
```

```
inline void VECTOR4D_ZERO(VECTOR4D_PTR v)
{(v)->x - (v)->y - (v)->z = 0.0; (v)->w = 1.0;}
```

```
// Инициализация векторов значениями компонентов
```

```
inline void VECTOR2D_INITXY(VECTOR2D_PTR v,
                             float x, float y)
{(v)->x = (x); (v)->y = (y);}
```

```
inline void VECTOR3D_INITXYZ(VECTOR3D_PTR v, float x,
                              float y, float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z);}
```

```
inline void VECTOR4D_INITXYZ(VECTOR4D_PTR v, float x,
                              float y, float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z); (v)->w = 1.0;}
```

```
// Инициализация векторов другими векторами
```

```
inline void VECTOR2D_INIT(VECTOR2D_PTR vdst,
                          VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }
```

```
inline void VECTOR3D_INIT(VECTOR3D_PTR vdst,
                          VECTOR3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
 (vdst)->z = (vsrc)->z; }
```

```
inline void VECTOR4D_INIT(VECTOR4D_PTR vdst,
                          VECTOR4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
 (vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }
```

```
// Копирование векторов
```

```
inline void VECTOR2D_COPY(VECTOR2D_PTR vdst,
                          VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }
```

```

inline void VECTOR3D_COPY(VECTOR3D_PTR vdst,
                          VECTOR3D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; }

inline void VECTOR4D_COPY(VECTOR4D_PTR vdst,
                          VECTOR4D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }

// Инициализация точек
inline void POINT2D_INIT(POINT2D_PTR vdst POINT2D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_INIT(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; }

inline void POINT4D_INIT(POINT4D_PTR vdst POINT4D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }

// Копирование точек
inline void POINT2D_COPY(POINT2D_PTR vdst POINT2D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_COPY(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; }

inline void POINT4D_COPY(POINT4D_PTR vdst, POINT4D_PTR vsrc)
{ (vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
  (vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }

```

Как видите, здесь имеются матрицы на все случаи жизни. Их даже слишком много — что приводит к избыточности, но кого это волнует?

## Матрицы

Далее следуют макросы для работы с матрицами. Здесь также имеются как макроопределения `tfdefine`, так и встраиваемые функции.

### НА ЗАМЕТКУ

При вызове функции `memset()` многие компиляторы используют побайтовое заполнение памяти. Этот процесс можно ускорить, используя заполнение машинными словами. Следует учесть, что в общем случае представление целых чисел отличается от представления тех же чисел, но в формате с плавающей точкой, т.е., например, представление `(float)5.0` отличается от представления `(int)5`; однако 32-битовое представление числа 0,0 с плавающей точкой эквивалентно представлению целого 32-битового числа 0, так что упомянутый трюк с использованием машинного слова при обнулении массивов вполне применим и для массивов чисел с плавающей точкой.

// Макросы для работы с матрицами

```

// Макросы для обнуления матриц
#define MAT_ZERO_2X2(m) {memset((void *)(m), 0, \
    sizeof(MATRIX2X2));}
#define MAT_ZERO_3X3(m) {memset((void *)(m), 0, \
    sizeof(MATRIX3X3));}
#define MAT_ZERO_4X4(m) {memset((void *)(m), 0, \
    sizeof(MATRIX4X4));}
#define MAT_ZERO_4X3(m) {memset((void *)(m), 0, \
    sizeof(MATRIX4X3));}

// Макросы для инициализации единичными матрицами
#define MAT_IDENTITY_2X2(m) {memcpy((void *)(m), \
    (void *)&IMAT_2X2, sizeof(MATRIX2X2));}
#define MAT_IDENTITY_3X3(m) {memcpy((void *)(m), \
    (void *)&IMAT_3X3, sizeof(MATRIX3X3));}
#define MAT_IDENTITY_4X4(m) {memcpy((void *)(m), \
    (void *)&IMAT_4X4, sizeof(MATRIX4X4));}
#define MAT_IDENTITY_4X3(m) {memcpy((void *)(m), \
    (void *)&IMAT_4X3, sizeof(MATRIX4X3));}

// Макросы копирования матриц
#define MAT_COPY_2X2(src_mat, dest_mat) \
    {memcpy((void *)(dest_mat), (void *)(src_mat), \
    sizeof(MATRIX2X2));}
#define MAT_COPY_3X3(src_mat, dest_mat) \
    {memcpy((void *)(dest_mat), (void *)(src_mat), \
    sizeof(MATRIX3X3));}
#define MAT_COPY_4X4(src_mat, dest_mat) \
    {memcpy((void *)(dest_mat), (void *)(src_mat), \
    sizeof(MATRIX4X4));}
#define MAT_COPY_4X3(src_mat, dest_mat) \
    {memcpy((void *)(dest_mat), (void *)(src_mat), \
    sizeof(MATRIX4X3));}

// Транспонирование матриц
inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m)
{
    MATRIX3X3 mt;
    mt.M00 = m->M00; mt.M01 = m->M10; mt.M02 = m->M20;
    mt.M10 = m->M01; mt.M11 = m->M11; mt.M12 = m->M21;
    mt.M20 = m->M02; mt.M21 = m->M12; mt.M22 = m->M22;
    memcpy((void *)m, (void *)&mt, sizeof(MATRIX3X3));
}

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m)
{
    MATRIX4X4 mt;
    mt.M00 = m->M00; mt.M01 = m->M10;
    mt.M02 = m->M20; mt.M03 = m->M30;
    mt.M10 = m->M01; mt.M11 = m->M11;
    mt.M12 = m->M21; mt.M13 = m->M31;
    mt.M20 = m->M02; mt.M21 = m->M12;
    mt.M22 = m->M22; mt.M23 = m->M32;
    mt.M30 = m->M03; mt.M31 = m->M13;
    mt.M32 = m->M23; mt.M33 = m->M33;
    memcpy((void *)m, (void *)&mt, sizeof(MATRIX4X4));
}

```

```

inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m,
                               MATRIX3X3_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10; mt->M02 = m->M20;
  mt->M10 = m->M01; mt->M11 = m->M11; mt->M12 = m->M21;
  mt->M20 = m->M02; mt->M21 = m->M12; mt->M22 = m->M22; }

```

```

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m,
                               MATRIX4X4_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10;
  mt->M02 = m->M20; mt->M03 = m->M30;
  mt->M10 = m->M01; mt->M11 = m->M11;
  mt->M12 = m->M21; mt->M13 = m->M31;
  mt->M20 = m->M02; mt->M21 = m->M12;
  mt->M22 = m->M22; mt->M23 = m->M32;
  mt->M30 = m->M03; mt->M31 = m->M13;
  mt->M32 = m->M23; mt->M33 = m->M33; }

```

// Следующие функции можно переписать как макросы, но при этом снизится надежность из-за отсутствия проверки типов

// Обмен столбцов

```

inline void MAT_COLUMN_SWAP_2X2(MATRIX2X2_PTR m, int c,
                                 MATRIX1X2_PTR v)
{ m->M[0][c] = v->M[0]; m->M[1][c] = v->M[1]; }

```

```

inline void MAT_COLUMN_SWAP_3X3(MATRIX3X3_PTR m, int c,
                                 MATRIX1X3_PTR v)
{ m->M[0][c] = v->M[0]; m->M[1][c] = v->M[1];
  m->M[2][c] = v->M[2]; }

```

```

inline void MAT_COLUMN_SWAP_4X4(MATRIX4X4_PTR m, int c,
                                 MATRIX1X4_PTR v)
{ m->M[0][c] = v->M[0]; m->M[1][c] = v->M[1];
  m->M[2][c] = v->M[2]; m->M[3][c] = v->M[3]; }

```

```

inline void MAT_COLUMN_SWAP_4X3(MATRIX4X3_PTR m, int c,
                                 MATRIX1X4_PTR v)
{ m->M[0][c] = v->M[0]; m->M[1][c] = v->M[1];
  m->M[2][c] = v->M[2]; m->M[3][c] = v->M[3]; }

```

Обратите внимание на то, что все функции для работы с матрицами используют указатели.

## Кватернионы

Следующий набор макросов предназначен для работы с кватернионами. Все эти функции являются встраиваемыми.

// Функции для работы с кватернионами

```

inline void QUAT_ZERO(QUAT_PTR q)
{ (q)->x = (q)->y = (q)->z = (q)->w = 0.0; }

```

```

inline void QUAT_INITWXYZ(QUAT_PTR q, float w, float x,
                          float y, float z)

```

```

{ (q)->w = (w); (q)->x = (x); (q)->y = (y); (q)->z = (z); }

inline void QUAT_INIT_VECTOR3D(QUAT_PTR q, VECTOR3D_PTR v)
{ (q)->w = 0; (q)->x = (v->x);
  (q)->y = (v->y); (q)->z = (v->z); }

inline void QUAT_INIT(QUAT_PTR qdst QUAT_PTR qsrc)
{ (qdst)->w = (qsrc)->w; (qdst)->x = (qsrc)->x;
  (qdst)->y = (qsrc)->y; (qdst)->z = (qsrc)->z; }

inline void QUAT_COPY(QUAT_PTR qdst QUAT_PTR qsrc)
{ (qdst)->x = (qsrc)->x; (qdst)->y = (qsrc)->y;
  (qdst)->z = (qsrc)->z; (qdst)->w = (qsrc)->w; }

```

Назначение этих функций очевидно как из названий, так и из кода — очень простого,

## Математика с фиксированной точкой

Последний набор макросов предназначен для преобразования чисел с фиксированной точкой и выделения целой и дробной частей этих чисел. Другие функции для работы с такими числами являются обычными, не встраиваемыми функциями. Дело в том, что в них используется встроенный ассемблер, который мне не хотелось использовать в заголовочном файле.

```

// Выделение целой и дробной частей числа с
// фиксированной точкой в формате 16.16
#define FIXP16_WP(fp) ((fp) >> FIXP16_SHIFT)
#define FIXP16_DP(fp) ((fp) && FIXP16_DP_MASK)

// Преобразование целых чисел и чисел с плавающей
// точкой в числа с фиксированной точкой в формате 16.16
#define INT_TO_FIXP16(i) ((i) << FIXP16_SHIFT)
#define FLOAT_TO_FIXP16(f) \
  (((float)(f) * (float)FIXP16_MAG+0.5))

// Преобразование числа с фиксированной точкой
// в число с плавающей точкой
#define FIXP16_TO_FLOAT(fp) ((float)fp)/FIXP16_MAG

```

## Прототипы

Теперь перейдем к прототипам функций математической библиотеки. Сейчас я просто перечислю эти функции; пояснения по поводу работы конкретных функций будут даны позже. Заметим, что здесь есть несколько функций из файлов `T3DLIB1.CPP`. Все функции разбиты на группы в соответствии с их функциональностью.

```

// Тригонометрические функции
float Fast_Sin(float theta);
float Fast_Cos(float theta);

// Функции для работы с расстояниями (из T3DLIB1.CPP)
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

```

```

// Полярные, цилиндрические и сферические функции
void POLAR2D_To_POINT2D(POLAR2D_PTR polar,
    POINT2D_PTR rect);
void POLAR2D_To_RectXY(POLAR2D_PTR polar,
    float Ч float *y);
void POINT2D_To_POLAR2D(POINT2D_PTR rect,
    POLAR2D_PTR polar);
void POINT2D_To_PolarRTh(POINT2D_PTR rect,
    float Ч float *theta);
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl,
    POINT3D_PTR rect);
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl,
    float Ч float *y, float *z);
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect,
    CYLINDRICAL3D_PTR cyl);
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect float *r,
    float *theta, float *z);
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph,
    POINT3D_PTR rect);
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph, float Ч
    float *y, float *z);
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect,
    SPHERICAL3D_PTR sph);
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect float *p,
    float *theta, float *phi);

// Функции для работы с двумерными векторами
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb,
    VECTOR2D_PTR vsum);
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb,
    VECTOR2D_PTR vdiff);
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va,
    VECTOR2D_PTR vscaled);
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);
void VECTOR2D_Build(VECTOR2D_PTR init, VECTOR2D_PTR term,
    VECTOR2D_PTR result);
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);

// Функции для работы с трехмерными векторами
void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb,
    VECTOR3D_PTR vsum);
VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb,
    VECTOR3D_PTR vdiff);
VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);

```

```

void VECTOR3D_Scale(float k, VECTOR3D_PTR va);
void VECTOR3D_Scale(float k, VECTOR3D_PTR va,
    VECTOR3D_PTR vscaled);
float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb,
    VECTOR3D_PTR vn);
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR3D_Length_Fast(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);
void VECTOR3D_Build(VECTOR3D_PTR init, VECTOR3D_PTR term,
    VECTOR3D_PTR result);
float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Print(VECTOR3D_PTR va, char *name);

// Функции для работы с четырехмерными векторами
void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb,
    VECTOR4D_PTR vsum);
VECTOR4D VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb,
    VECTOR4D_PTR vdiff);
VECTOR4D VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va,
    VECTOR4D_PTR vscaled);
float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb,
    VECTOR4D_PTR vn);
VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
float VECTOR4D_Length(VECTOR4D_PTR va);
float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term,
    VECTOR4D_PTR result);
float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Print(VECTOR4D_PTR va, char *name);

// Функции для работы с матрицами 2x2 (из T3DLIB1.CPP/H)
void Mat_Init_2X2(MATRIX2X2_PTR ma,
    float m00, float m01, float m10, float m11);
void Print_Mat_2X2(MATRIX2X2_PTR ma, char *name);
float Mat_Det_2X2(MATRIX2X2_PTR m);
void Mat_Add_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb,
    MATRIX2X2_PTR msum);
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb,
    MATRIX2X2_PTR mprod);
int Mat_Inverse_2X2(MATRIX2X2_PTR m, MATRIX2X2_PTR mi);
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X,
    MATRIX1X2_PTR B);

// Функции для работы с матрицами 3x3

```

```

// (часть из T3DLIB1.CPP|H)
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma, MATRIX3X2_PTR mb,
    MATRIX1X2_PTR mprod);
int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma, MATRIX3X3_PTR mb,
    MATRIX1X3_PTR mprod);
int Mat_Mul_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb,
    MATRIX3X3_PTR mprod);
inline int Mat_Init_3X2(MATRIX3X2_PTR ma, float m00,
    float m01, float m10, float m11,
    float m20, float m21);
void Mat_Add_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb,
    MATRIX3X3_PTR msum);
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb,
    VECTOR3D_PTR vprod);
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
void Mat_Init_3X3(MATRIX3X3_PTR ma, float m00, float m01,
    float m02, float m10, float m11, float m12,
    float m20, float m21, float m22);
void Print_Mat_3X3(MATRIX3X3_PTR ma, char *name);
float Mat_Det_3X3(MATRIX3X3_PTR m);
int Solve_3X3_System(MATRIX3X3_PTR A, MATRIX1X3_PTR X,
    MATRIX1X3_PTR B);

// Функции для работы с матрицами 4x4
void Mat_Add_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb,
    MATRIX4X4_PTR msum);
void Mat_Mul_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb,
    MATRIX4X4_PTR mprod);
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma, MATRIX4X4_PTR mb,
    MATRIX1X4_PTR mprod);
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va, MATRIX4X4_PTR mb,
    VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR3D_4X3(VECTOR3D_PTR va, MATRIX4X3_PTR mb,
    VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR4D_4X4(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
    VECTOR4D_PTR vprod);
void Mat_Mul_VECTOR4D_4X3(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
    VECTOR4D_PTR vprod);
int Mat_Inverse_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mi);
void Mat_Init_4X4(MATRIX4X4_PTR ma, float m00, float m01,
    float m02, float m03, float m10, float m11,
    float m12, float m13, float m20, float m21,
    float m22, float m23, float m30, float m31,
    float m32, float m33);
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);

// Функции для работы с кватернионами
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
void QUAT_Scale(QUAT_PTR q, float scale);
float QUAT_Norm(QUAT_PTR q);

```

```

float QUAT_Norm2(QUAT_PTR q);
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
void QUAT_Normalize(QUAT_PTR q);
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Unit_Inverse(QUAT_PTR q);
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Inverse(QUAT_PTR q);
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2,
    QUAT_PTR q3, QUAT_PTR qprod);
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v,
    float theta);
void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v,
    float theta);
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z,
    float theta_y, float theta_x);
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v,
    float *theta);
void QUAT_Print(QUAT_PTR q, char *name);

// Работа с двумерными параметрическими прямыми
void Init_Parm_Line2D(POINT2D_PTR p_init, POINT2D_PTR p_term,
    PARMLINE2D_PTR p);
void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t,
    POINT2D_PTR pt);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1,
    PARMLINE2D_PTR p2, float *t1,
    float *t2);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1,
    PARMLINE2D_PTR p2,
    POINT2D_PTR pt);

// Работа с трехмерными параметрическими прямыми
void Init_Parm_Line3D(POINT3D_PTR p_init,
    POINT3D_PTR p_term, PARMLINE3D_PTR p);
void Compute_Parm_Line3D(PARMLINE3D_PTR p, float t,
    POINT3D_PTR pt);

// Работа с плоскостями в трехмерном пространстве
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
    VECTOR3D_PTR normal, int normalize);
float Compute_Point_In_Plane3D(POINT3D_PTR pt,
    PLANE3D_PTR plane);
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline,
    PLANE3D_PTR plane,
    float *t, POINT3D_PTR pt);

// Функции для работы с числами с фиксированной точкой
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
FIXP16 FIXP16_DIV(FIXP16 fpl, FIXP16 fp2);
void FIXP16_Print(FIXP16 fp);

```

**Немало функций, не правда ли? Как я уже говорил, наша библиотека может практически все, что только может понадобиться при разработке трехмерных игр.**

## Глобальные переменные

В нашей математической библиотеке глобальных переменных очень мало, причем все они — из упоминавшихся ранее файлов `T3DLIB1.CPP|H`. Это — таблицы поиска для быстрого вычисления значений синусов и косинусов,

```
// Таблицы поиска для вычисления тригонометрических функций
extern float cos_look[361];
extern float sin_look[361];
```

Обратите внимание, что данные таблицы содержат значения функций для углов, выраженных в градусах, а не радианах. Таблицы содержат по 361 запись, для углов от 0° до 360° включительно. Да, конечно, 360° — это то же, что и 0°, но наличие такой записи в таблице облегчает реализацию некоторых алгоритмов. Вам только надо не забыть инициализировать таблицы при помощи вызова `Build_Sin_Cos_Tables()` в начале вашего приложения.



В более надежной и интеллектуальной математической библиотеке может иметься как гораздо большее число глобальных переменных, отслеживающих состояние библиотеки, так и более сложные структуры данных.

## API математической библиотеки

А сейчас начинается самая интересная часть этой главы. Вы узнаете, что именно делает каждая из функций математической библиотеки, причем этот материал снабжен многочисленными демонстрационными примерами. Кроме того, я покажу вам "внутренности" некоторых функций, чтобы вы могли увидеть, как разрабатываются такие функции. Описания функций, которые являются частью математической поддержки библиотеки `T3DLIB1.CPP|H`, даны в главе 3, "Виртуальный компьютер для программирования трехмерных игр".

## Тригонометрические функции

### Прототипы функций

```
float Fast_Sin(float theta);
float Fast_Cos(float theta);
```

### Исходный текст функции

```
float Fast_Sin(float theta)
{
    // Функция использует таблицу sin_look[] для поиска
    // значения синуса. Обрабатывает отрицательные значения
    // углов. Для дробных значений выполняется интерполяция
    theta = fmodf(theta, 360);

    // Делаем угол положительным
    if (theta < 0) theta += 360.0;
    // Вычисляем целую и дробную часть для интерполяции
    int theta_int = (int)theta;
    float theta_frac = theta - theta_int;

    // Обратите внимание на корректность обработки угла 359
    // градусов из-за наличия в таблице угла 360 градусов
```

```

return (sin_look[theta_int] +
        theta_frac*(sin_look[theta_int+1] -
                    sin_look[theta_int]));
} // Fast_Sin

```

### Назначение

Функции `Fast_Sin()` и `Fast_Cos()` вычисляют синус и косинус переданного в качестве параметра угла с использованием таблиц поиска и линейной **интерполяции** для получения **уточненного** значения. Данные функции работают быстрее соответствующих встроенных функций математической библиотеки C/C++. Передаваемый параметр должен представлять собой величину угла в градусах. На рис. 5.11 проиллюстрировано применение интерполяции для вычисления тригонометрических функций.

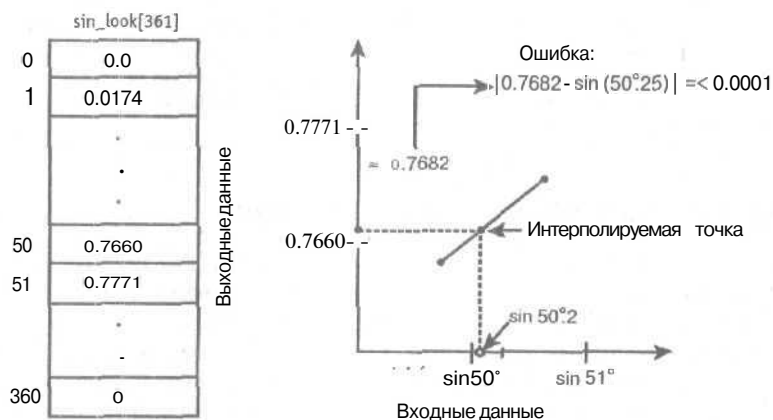


Рис. 5. И. Использование интерполяции в функциях `Fast_*`

### Пример использования

```

// Инициализация таблиц
Build_Sin_Cos_Tables();

// Вычисление синуса и косинуса угла 50.2 градуса
float answer1 = Fast_Sin(50.2);
float answer2 = Fast_Cos(50.2);

```

## Функции для работы с системами координат

Все функции из данного раздела не возвращают никаких значений.

### Прототип функции

```

void POLAR2D_To_POINT2D(POLAR2D_PTR polar,
                        POINT2D_PTR rect);

```

### Назначение

Данная функция преобразует двумерные полярные координаты в декартовы.

### Пример использования

```

POLAR2D pp = {5, PI/2};
POINT2D pr = {0,0};

```

```
// Преобразование полярных координат в декартовы
POLAR2D_To_POINT2D(&pp, &pr);
```

#### **Прототип функции**

```
void POLAR2D_To_RectXY(POLAR2D_PTR polar,
    float *x, float *y);
```

#### **Назначение**

Данная функция преобразует двумерные полярные координаты в явные значения декартовых координат.

#### **Пример использования**

```
POLAR2D pp = {5, PI/2};
float x=0, y=0;
```

```
// Преобразование полярных координат в декартовы
POLAR2D_To_RectXY(&pp, &x, &y)
```

#### **Прототип функции**

```
void POINT2D_To_POLAR2D(POINT2D_PTR rect
    POLAR2D_PTR polar);
```

#### **Назначение**

Функция преобразует двумерные декартовы координаты в полярные.

#### **Пример использования**

```
POLAR2D pp = {0, 0};
POINT2D pr = {10, 20};
```

```
// Преобразование декартовых координат в полярные
POINT2D_To_POLAR2D(&pr, &pp);
```

#### **Прототип функции**

```
void POINT2D_To_PolarRTh(POINT2D_PTR rect
    float *r, float *theta);
```

#### **Назначение**

Функция преобразует двумерные декартовы координаты в явные значения полярных координат.

#### **Пример использования**

```
POLAR2D pp = {3, PI};
float r=0, theta=0;
```

```
// Преобразование декартовых координат в полярные
POINT2D_To_PolarRTh(&pp, &r, &theta);
```

#### **Прототип функции**

```
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl,
    POINT3D_PTR rect);
```

#### **Назначение**

Данная функция преобразует цилиндрические координаты в декартовы.

#### **Пример использования**

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r, theta, z
POINT3D pr = {0, 0, 0};
```

```
// Преобразование цилиндрических координат в декартовы
CYLINDRICAL3D_To_POINT3D(&pc, &pr);
```

#### **Прототип функции**

```
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl,
    float *x, float *y,
    float *z);
```

#### **Назначение**

Данная функция преобразует цилиндрические координаты в явные значения декартовых координат.

#### **Пример использования**

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r, theta, z
float x=0, y=0, z=0;
```

```
// Преобразование цилиндрических координат в декартовы
CYLINDRICAL3D_To_RectXYZ(&pc, &x, &y, &z);
```

#### **Прототип функции**

```
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect,
    CYLINDRICAL3D_PTR cyl);
```

#### **Назначение**

Функция преобразует декартовы координаты в цилиндрические.

#### **Пример использования**

```
CYLINDRICAL3D pc = {0,0,0}; // r, theta, z
POINT3D pr = {1,2,3};
```

```
// Преобразование декартовых координат в цилиндрические
POINT3D_To_CYLINDRICAL3D(&pr, &pc);
```

#### **Прототип функции**

```
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect, float *r,
    float *theta, float *z);
```

#### **Назначение**

Функция преобразует декартовы координаты в явные значения цилиндрических координат.

#### **Пример использования**

```
POINT3D pr = {1,2,3};
float r=0, theta=0, z=0;
```

```
// Преобразование декартовых координат в цилиндрические
POINT3D_To_CylindricalRThZ(&pr, &r, &theta, &z);
```

#### **Прототип функции**

```
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph,
    POINT3D_PTR rect);
```

#### **Назначение**

Функция преобразует сферические координаты в декартовы.

#### **Пример использования**

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
POINT3D pr = {0,0,0};
```

```
// Преобразование сферических координат в декартовы
SPHERICAL3D_To_POINT3D(&ps, &pr);
```

#### Прототип функции

```
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph, float *x,
float *y, float *z);
```

#### Назначение

Функция преобразует сферические координаты в явные значения декартовых координат.

#### Пример использования

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
float x=0, y=0, z=0;
```

```
// Преобразование сферических координат в декартовы
SPHERICAL3D_To_RectXYZ(&ps, ax, &y, &z);
```

#### Прототип функции

```
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect
SPHERICAL3D_PTR sph);
```

#### Назначение

Функция преобразует декартовы координаты в сферические.

#### Пример использования

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
POINT3D pr = {0,0,0};
```

```
// Преобразование декартовых координат в сферические
POINT3D_To_SPHERICAL3D(&pr, &ps);
```

#### Прототип функции

```
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect float *p,
float *theta, float *phi);
```

#### Назначение

Функция преобразует декартовы координаты в явные значения сферических координат.

#### Пример использования

```
POINT3D pr = {10,20,30};
float p=0, theta=0, phi=0;
```

```
// Преобразование декартовых координат в сферические
POINT3D_To_SphericalPThPh(&pr, &p, &theta, &phi);
```

## Функции для работы с векторами

Поддержка векторов в математической библиотеке включает двух-, трех- и четырехмерные векторы. Четырехмерные векторы в данном случае представляют собой векторы в однородных координатах. Например, если мы рассматриваем четырехмерный вектор  $p = \langle x, y, z, w \rangle$ , то он преобразуется в реальный трехмерный вектор  $p' = \langle x/w, y/w, z/w \rangle$ .



Заметим, что во многих операциях с четырехмерными векторами компонент  $w$  рассматривается как фиктивный и работа осуществляется только с компонентами  $x$ ,  $y$  и  $z$ .

Ясно, что не все операции доступны для векторов всех типов. Например, векторное произведение двумерных векторов лишено смысла. Однако большинство рассматриваемых здесь функций имеют версии для работы с векторами всех размерностей, и в качестве примера приводится работа только одной из этих функций.

Большинство функций возвращают значения, которые являются числами с плавающей точкой либо векторами. Последнее, очевидное замечание — по имени функции сразу понятно, для векторов какой размерности она предназначена: наличие 2D говорит о том, что это функция для двумерных векторов, 3D — трехмерных и 4D — четырехмерных.

#### НА ЗАМЕТКУ

Типы данных для векторов и точек по сути одинаковы. Таким образом, везде, где используется тип `VECTOR#D`, можно использовать тип `POINT#D`. Понятно, однако, что некоторые вычисления над точками не имеют никакого смысла, например, векторное произведение. Но при построении вектора можно использовать как две точки, так и пару векторов или точку и вектор.

### Прототипы функций

```
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb,
    VECTOR2D_PTR vsum);
void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb,
    VECTOR3D_PTR vsum);
void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb,
    VECTOR4D_PTR vsum);
```

### Назначение

Функции `VECTOR#D_Add()` суммируют переданные в качестве параметров векторы, как показано на рис. 5.12, и возвращают полученный в результате вектор.

### Пример использования

```
VECTOR3D v1 = {1,2,3};
VECTOR3D v2 = {5,6,7};
VECTOR3D vsum; // Для хранения результата

// Суммирование векторов
VECTOR3D_Add(&v1, &v2, &vsum);
```

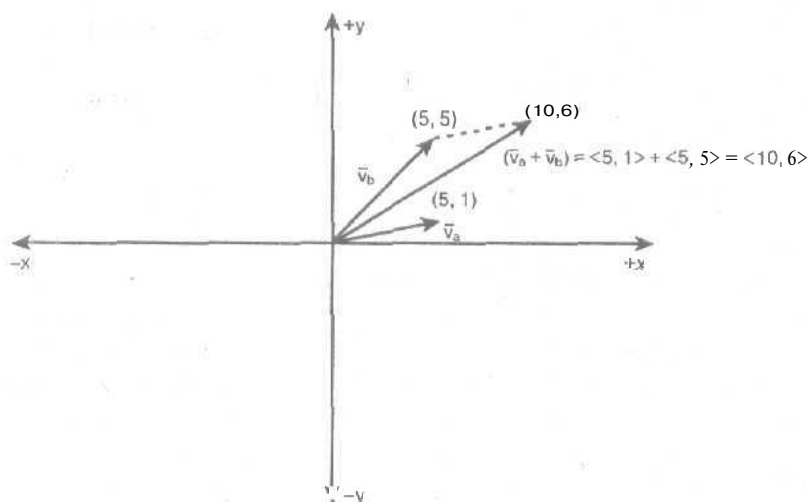


Рис. 5.12. Сложение векторов

### Прототипы функций

```
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);  
VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);  
VECTOR4D VECTOR3D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

### Назначение

Функции `VECTOR*D_VECTOR*D_Add()` суммируют переданные в качестве параметров векторы и передают суммарный вектор в виде возвращаемого значения в стек.

### Пример использования

```
VECTOR2D v1 = {1,2};  
VECTOR2D v2 = {5,6};  
  
// Суммирование векторов  
VECTOR2D vsum = VECTOR2D_Add(&v1, &v2);
```

### Прототипы функций

```
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb,  
    VECTOR2D_PTR vdiff);  
void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb,  
    VECTOR3D_PTR vdiff);  
void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb,  
    VECTOR4D_PTR vdiff);
```

### Назначение

Функции `VECTOR*D_Sub()` вычисляют разность переданных в качестве параметров векторов, как показано на рис. 5.13, и возвращают полученный в результате вектор.

### Пример использования

```
VECTOR3D v1 = {1,2,3};  
VECTOR3D v2 = {5,6,7};  
VECTOR3D vdiff; // Для хранения результата  
  
// Вычитание векторов  
VECTOR3D_Sub(&v1, &v2, &vdiff);
```

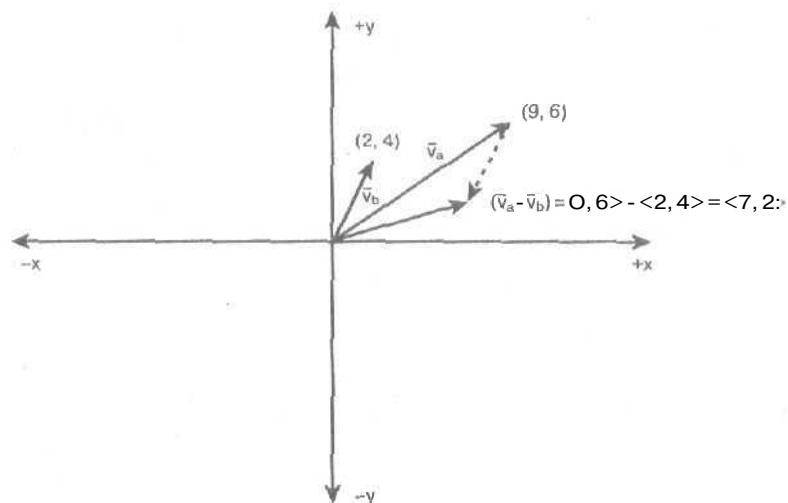


Рис. 5.13. Вычитание векторов

### Прототипы функций

```
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);  
VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);  
VECTOR4D VECTOR3D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

### Назначение

Функции `VECTOR*D VECTOR*_D_Sub()` вычисляют разность переданных в качестве параметров векторов и передают результат в виде возвращаемого значения в стек.

### Пример использования

```
VECTOR2D v1 = {1,2};  
VECTOR2D v2 = {5,6};  
  
// Вычитание векторов  
VECTOR2D vdiff = VECTOR2D_Sub(&v1, &v2);
```

### Прототипы функций

```
void VECTOR2D_Scale(float k, VECTOR2D_PTR va,  
    VECTOR2D_PTR vscaled);  
void VECTOR3D_Scale(float k, VECTOR3D_PTR va,  
    VECTOR3D_PTR vscaled);  
void VECTOR4D_Scale(float k, VECTOR4D_PTR va,  
    VECTOR4D_PTR vscaled);
```

### Назначение

Функции `void VECTOR*_D_Scale()` масштабируют переданный в качестве параметра `va` вектор (рис. 5.14), увеличивая его в  $k$  раз, и возвращают масштабированный вектор как `vscaled`.

### Пример использования

```
VECTOR3D v1 = {1,1,1};  
VECTOR3D vs;  
  
// Увеличение в 50 раз  
VECTOR3D_Scale(50, &v1, &vs);  
// Изменение направления на обратное  
VECTOR3D_Scale(-1, &v1, &vs);
```

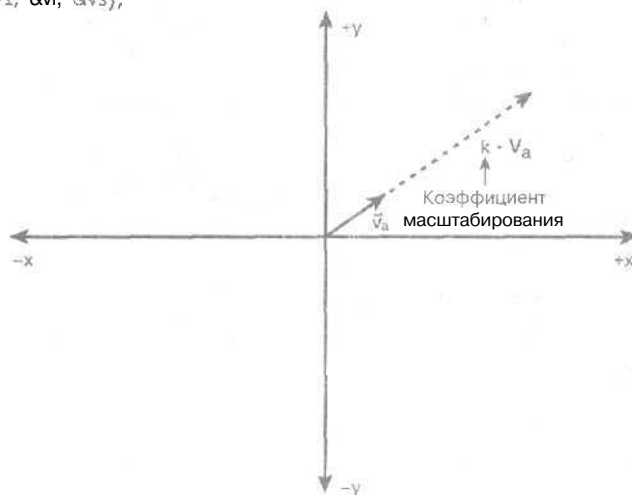


Рис. 5.14. Масштабирование вектора

### Прототипы функций

```
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);  
float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);  
float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

### Назначение

Функции float VECTOR\*D\_Dot() возвращают результат скалярного произведения двух векторов. В четырехмерной версии компонента w в вычислении не участвует.

### Пример использования

```
VECTOR4D v1={1,0,0,1}; // Единичный вектор вдоль оси x  
VECTOR4D v2={0,1,0,1}; // Единичный вектор вдоль оси y
```

```
// Скалярное произведение равно 0!  
float dp=VECTOR4D_Dot(&v1, &v2);
```

### Прототипы функций

```
void VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb,  
                    VECTOR3D_PTR vn);  
void VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb,  
                    VECTOR4D_PTR vn);
```

### Назначение

Функции void VECTOR\*D\_Cross() вычисляют векторное произведение переданных и качестве параметров векторов (рис. 5.15) и сохраняют результат по адресу vn. В четырехмерной версии компонента w в вычислении не участвует.

### Пример использования

```
VECTOR3D vx = {1,0,0};  
VECTOR3D vy = {0,1,0};  
VECTOR3D vcross;
```

```
// Векторное произведение двух векторов, направленных  
// вдоль осей x и y, дает в результате вектор,  
// направленный вдоль оси z  
VECTOR3D_Cross(&vx, &vy, &vcross);
```

### Прототипы функций

```
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);  
VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

### Назначение

Функции VECTOR\*D VECTOR\*D\_Cross() вычисляют векторное произведение переданных в качестве параметров векторов (рис. 5.15), и возвращают вычисленное значение в стеке. В четырехмерной версии компонента w в вычислении не участвует.

### Пример использования

```
VECTOR3D vx = {1,0,0};  
VECTOR3D vy = {0,1,0};
```

```
// Векторное произведение двух векторов, направленных  
// вдоль осей x и y, дает в результате вектор,  
// направленный вдоль оси z  
VECTOR3D vcross = VECTOR3D_Cross(&vx, &vy);
```

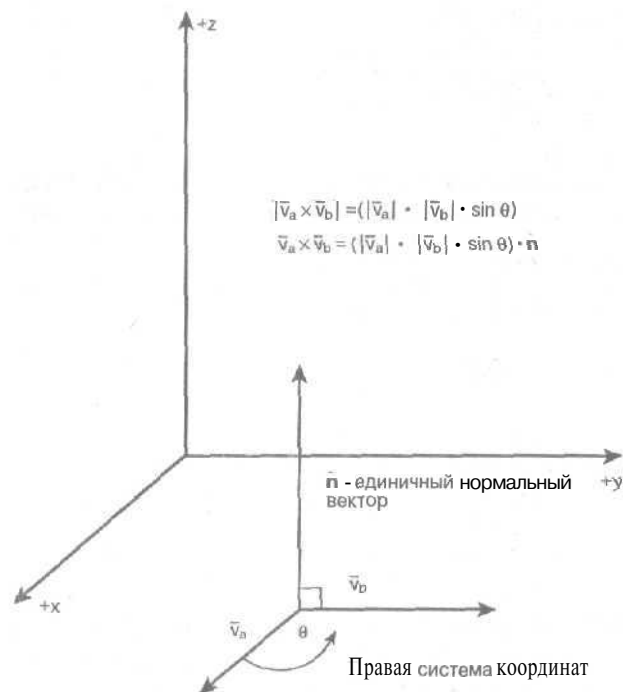


Рис. 5.15. Векторное произведение

#### Прототипы функций

```
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR4D_Length(VECTOR4D_PTR va);
```

#### Назначение

Функция float VECTOR\*D\_Length() вычисляет длину вектора как квадратный корень из суммы квадратов его компонентов.

#### Пример использования

```
VECTOR2D v1={3,4};
```

```
// Длина вектора равна 5
float length = VECTOR3D_Length(&v1);
```

#### Прототипы функций

```
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);
float VECTOR3D_Length_Fast(VECTOR3D_PTR va);
float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
```

#### Назначение

Функция float VECTOR\*D\_Length\_Fast() вычисляет длину вектора с использованием приближения рядом Тейлора. Погрешность составляет несколько процентов, но скорость вычисления на порядок выше, чем у предыдущих функций.

#### Пример использования

```
VECTOR2D v1={3,4};
```

```
// Длина вектора должна быть равна 5
float length = VECTOR3D_Length_Fast(&v1);
```

### Прототипы функций

```
void VECTOR2D_Normalize(VECTOR2D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va);
```

### Назначение

Функции `void VECTOR*D_Normalize()` нормализуют передаваемые в качестве параметров векторы путем деления компонент вектора на его длину. Обратите внимание, что исходный вектор при вызове функции изменяется.

### Пример использования

```
VECTOR2D v={5,6};

// Нормализация v, v = v/|v|
VECTOR2D_Normalize(&v);
```

### Прототипы функций

```
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);
void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);
void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
```

### Назначение

Функции `void VECTOR*D_Normalize()` нормализуют вектор `va` и сохраняют полученное значение в переменной `vn`.

### Пример использования

```
VECTOR3D v={1,2,3};
VECTOR3D vn;

// vn = v/|vn|
VECTOR3D_Normalize(&v, &vn);
```

### Прототипы функций

```
void VECTOR2D_Build(VECTOR2D_PTR init VECTOR2D_PTR term,
    VECTOR2D_PTR result);
void VECTOR3D_Build(VECTOR3D_PTR init VECTOR3D_PTR term,
    VECTOR3D_PTR result);
void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term,
    VECTOR4D_PTR result);
```

### Назначение

Функции `void VECTOR*D_Build()` строят вектор `init→term` и сохраняют его в переменной `result`. Это хороший пример функций, которые могут работать как с векторами, так и с точками — для создания вектора, определяемого двумя точками.

### Пример использования

```
POINT3D p1={1,2,3}, p2={4,5,6};
VECTOR3D v12; // Для хранения результата
```

```
// Создаем вектор из точки p1 в точку p2
VECTOR3D_Build(&p1, &p2, &v);
```

Обратите **внимание** на то, что нам не надо использовать приведение типов, поскольку точки и векторы одинаковой размерности принадлежат к одному типу.

### Прототипы функций

```
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);
float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);
float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

### Назначение

Функции `float VECTOR*D_CosTh()` вычисляют косинус угла между двумя векторами (рис. 5.16).

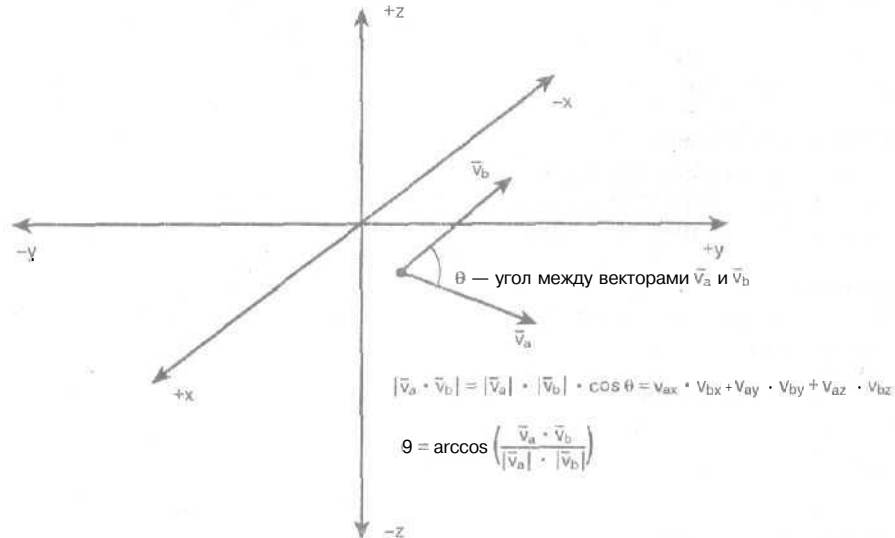


Рис. 5.16. Скалярное произведение и угол между векторами

В ряде трехмерных алгоритмов формула скалярного произведения векторов

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos \theta$$

используется для вычисления угла  $\theta$  между векторами:

$$\theta = \arccos \left( \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| |\mathbf{v}|} \right)$$

Однако зачастую нам достаточно знать значение косинуса угла между векторами, а не сам угол. Именно эту величину — косинус угла между двумя векторами — и вычисляют рассматриваемые функции.

### Пример использования

```
VECTOR3D u = {0,1,0};
VECTOR3D v = {1,1,0};
```

```
// Поиск косинуса угла между u и v
VECTOR3D_CosTh(&u, &v);
```

**СОВЕТ**

Это достаточно распространенная практика в программировании игр — использовать не конкретное значение, а некоторую функцию от него, если это приводит к упрощению и повышению скорости вычислений. Например, проверяя, попадает ли некоторый объект с координатами  $(x, y, z)$  в область радиуса  $r$  от начала координат, можно вычислить значение  $\sqrt{x^2 + y^2 + z^2}$  и сравнивать его со значением  $r$ . Однако проще и быстрее будет заранее вычислить  $r^2$  и сравнивать с ним величину  $x^2 + y^2 + z^2$ .

**НА ЗАМЕТКУ**

Эта функция очень полезна при удалении скрытых поверхностей, когда нам надо определить угол между направлением просмотра и нормалью к поверхности.

**Прототипы функций**

```
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);
void VECTOR3D_Print(VECTOR3D_PTR va, char *name);
void VECTOR4D_Print(VECTOR4D_PTR va, char *name);
```

**Назначение**

Функции `void VECTOR*D_Print()` выводят значение компонентов вектора в удобочитаемом виде, вместе с именем вектора, представленном параметром `name`. Используется в отладочных целях; вывод направляется в файл, открытый с помощью функции `Open_Error_File()`.

**Пример использования**

```
// Открываем файл
Open_Error_File("error.txt");
VECTOR3D v={1,2,3};

// Выводим вектор
VECTOR3D_Print(&v,"Velocity");

// Закрываем файл
Close_Error_File();
```

Содержимое файла ERROR.TXT:

```
Opening Error Output File (error.txt) on
Sat Jan 29 22:36:00.520, 2000
Velocity=[ 1.000000, 2.000000, 3.000000, ]
Closing Error Output File.
```

**Функции для работы с матрицами**

Очередной набор функций предназначен для выполнения преобразований и математических операций с матрицами. В основном при разработке трехмерных игр используются матрицы  $3 \times 3$  и  $4 \times 4$ , но иногда мы будем “химичить” и работать с матрицами  $4 \times 3$ , считая последний, фиктивный столбец таких матриц имеющим предопределенное значение. Впрочем, как бы я ни пытался учесть все случаи преобразований или умножения матриц, какие-то варианты окажутся пропущенными — однако дописать нужные функции и внести их в библиотеку достаточно просто. И последнее замечание — в библиотеке T3DLIB1.CPP имеются некоторые функции для работы с матрицами  $2 \times 2$ .

Хотя функции для работы с матрицами и похожи на функции для работы с векторами в том, что имеются варианты функций для выполнения одинаковых действий с матрицами разных размерностей, здесь я не стал объединять их в группы с одинаковой функциональностью. Это объясняется тем, что иногда функции для более высоких размерностей реализованы не так, как для низких. Поэтому мы будем сначала рассматривать функции для работы с матрицами 2x2, затем — 3x3 и 4x4.

### Прототип функции

```
void Mat_Init_2X2(MATRIX2X2_PTR ma,
    float m00, float m01,
    float m10, float m11);
```

### Назначение

Функция `void Mat_Init_2X2()` инициализирует матрицу `ma` значениями с плавающей точкой (передаваемыми построчно).

### Пример использования

```
MATRIX2X2 ta;

// Инициализация единичной матрицы
Mat_Init_2X2(&ta, 1, 0, 0, 1);
```

### Прототип функции

```
void Mat_Add_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb,
    MATRIX2X2_PTR msum);
```

### Назначение

Функция `void Mat_Add_2X2()` суммирует матрицы `(ma+mb)` и сохраняет результат в матрице `msum`.

### Пример использования

```
MATRIX2X2 m1 = {1,2, 3,4};
MATRIX2X2 m2 = {4,9, 5,6};
MATRIX2X2 msum;

// Сложение матриц
Mat_Add_2X2(&m1, &m2, &msum);
```

### Прототип функции

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb,
    MATRIX2X2_PTR mprod);
```

### Исходный текст функции

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb,
    MATRIX2X2_PTR mprod)
{
    // Умножение двух матриц 2x2
    mprod->M00 = ma->M00*mb->M00 + ma->M01*mb->M10;
    mprod->M01 = ma->M00*mb->M01 + ma->M01*mb->M11;
    mprod->M10 = ma->M10*mb->M00 + ma->M11*mb->M10;
    mprod->M11 = ma->M10*mb->M01 + ma->M11*mb->M11;
} // Mat_Mul_2X2
```

### Назначение

Функция `void Mat_Mul_2X2()` умножает матрицы `(ma*mb)` и сохраняет полученный результат в матрице `mprod`. Обратите внимание, что для повышения скорости матрицы перемножаются "в лоб", без использования циклов.

### Пример использования

```
MATRIX2X2 m1 = {1,2, 3,4};  
MATRIX2X2 m2 = {1,0, 0,1};  
MATRIX2X2 mprod;
```

```
// Умножение; заметим, что m1*m2 = m1, т.к. m2=I  
Mat_Mul_2X2(&m1,&m2, fcmprod);
```

### Прототип функции (из T3DLIB1.CPP|H)

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma, MATRIX3X2_PTR mb,  
MATRIX1X2_PTR mprod);
```

### Исходный текст функции

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma, MATRIX3X2_PTR mb,  
MATRIX1X2_PTR mprod)
```

```
{  
    // Функция перемножает матрицу 1x2 на матрицу 3x2  
    // Используется фиктивный элемент для того, чтобы  
    // матрицу 1x2 к виду 1x3 для корректности умножения  
    for (int col=0; col<2; col++)  
    {  
        // Вычисляем скалярное произведение  
        // строки ma и столбца mb  
        float sum = 0; //Хранилище результата  
        for (int index=0; index<2; index++)  
        {  
            // Сложение произведений  
            sum += (ma->M[index]*mb->M[index][col]);  
        } // for index  
        // Последний элемент умножаем на 1  
        sum += mb->M[index][col];  
        // Вставка полученного элемента в результат  
        mprod->M[col] = sum;  
    } // for col  
    return(1);  
} // Mat_Mul_1X2_3X2
```

### Назначение

Функция `int Mat_Mul_1X2_3X2()` представляет собой специализированную функцию для умножения матрицы 1x2 (представляющую в основном двумерные точки) на матрицу 3x2, которая представляет поворот и перенос. Для корректности операции требуется добавление фиктивного элемента, чтобы внутренние размерности матриц стали одинаковыми (рис. 5.17).

### Пример использования

```
MATRIX1X2 p1={5,5}, p2; //Точка
```

```
// Поворот и перенос  
MATRIX3X2 m = {cos(th), sin(th),  
               -sin(th), cos(th),  
               dx,   dy};
```

```
// Умножение матриц  
Mat_Mul_1X2_3X2(&p1,&m,&p2);
```

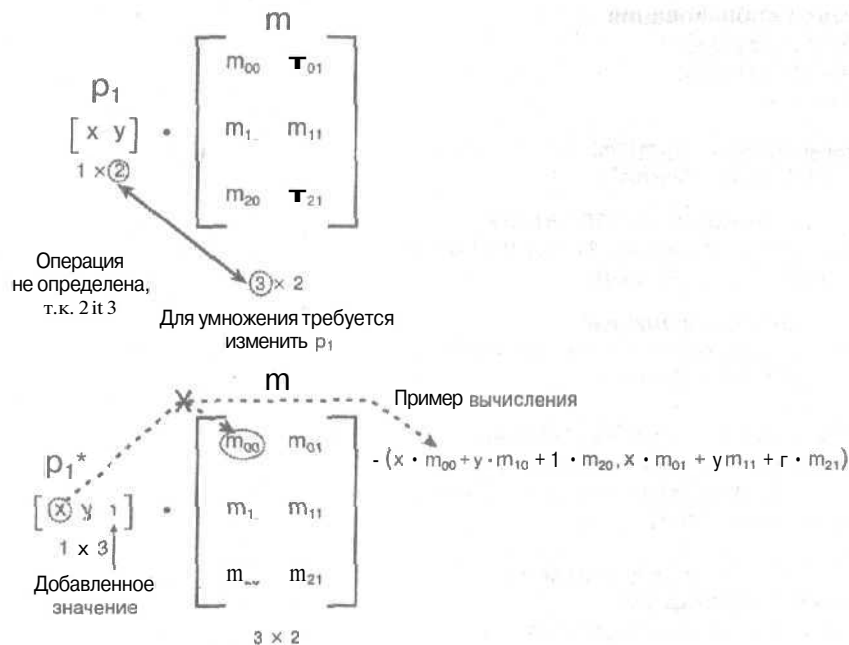


Рис. 5.17. Выполнение умножения матриц  $1 \times 2$  и  $3 \times 2$

#### СОВЕТ

Приведенную функцию можно ускорить, развернув циклы. Однако код с использованием циклов понятнее в силу итеративности решаемой задачи. Запомните — никогда не приступайте к оптимизации, пока это не является совершенно необходимым. Приступив к оптимизации, начинайте с оптимизации алгоритма. Лишь когда возможности оптимизации алгоритма исчерпаны, можно приступить к оптимизации кода, и только исчерпав возможности оптимизации кода на высоком уровне, можно использовать для оптимизации ассемблерные вставки.

#### Прототип функции

```
float Mat_Det_2X2(MATRIX2X2_PTR m);
```

#### Назначение

Функция `float Mat_Det_2X2()` вычисляет и возвращает значение определителя матрицы  $2 \times 2$ .

#### Пример использования

```
MATRIX2x2 m = {1,2,4,8}; // Матрица сингулярна
```

```
// Вычисляем определитель
float det = Mat_Det_2X2(&m);
```

#### Прототип функции

```
int Mat_Inverse_2X2(MATRIX2X2_PTR m, MATRIX2X2_PTR mi);
```

#### Назначение

Функция `int Mat_Inverse_2X2()` вычисляет матрицу, обратную матрице  $t$ , и сохраняет ее в матрице  $mi$ . Если обратная матрица существует, функция возвращает 1; в противном случае функция возвращает 0, а матрица  $mi$  является неопределенной.

#### Пример использования

```
MATRIX2X2 mA = {1,2, 4, -3};  
MATRIX2X2 mI;
```

```
// Вычисление матрицы, обратной A  
if (Mat_Inverse_2X2(&mA, &mI))  
{  
    // Обратная матрица существует...  
} // if  
else  
{  
    // Обратная матрица не существует...  
} // else
```

#### Прототип функции

```
void Print_Mat_2X2(MATRIX2X2_PTR ma, char *name);
```

#### Назначение

Функция `void Print_Mat_2X2()` выводит матрицу в удобочитаемом формате вместе с именем, переданным в строке `name`. Вывод осуществляется в файл, открытый при помощи вызова функции `Open_Error_File()`.

#### Пример использования

```
MATRIX2X2 m - {1,2,3,4};  
  
// Выводим матрицу на экран  
Open_Error_File("", stdout);  
Print_Mat_2X2(&m, "Matrix m");  
// Закрываем файл  
Close_Error_File();
```

#### Прототип функции

```
void Mat_Init_3X3(MATRIX3X3_PTR ma,  
                 float m00, float m01, float m02,  
                 float m10, float m11, float m12,  
                 float m20, float m21, float m22);
```

#### Назначение

Функция `void Mat_Init_3X3()` инициализирует матрицу `ma` непосредственными значениями элементов, передаваемых построчно.

#### Пример использования

```
MATRIX3X3 та;  
  
// Инициализируем единичную матрицу  
Mat_Init_3X3(&та, 1,0,0, 0,1,0, 0,0,1);
```

#### Прототип функции (из T3DLIB1.CPP|H)

```
inline int Mat_Init_3X2(MATRIX3X2_PTR ma,  
                       float m00, float m01,  
                       float m10, float m11,  
                       float m20, float m21);
```

#### Назначение

Функция `void Mat_Init_3X2()` инициализирует матрицу `ma` непосредственными значениями элементов, передаваемых построчно.

### Пример использования

```
MATRIX3X2 ma;
```

```
// Верхний левый угол делаем единичной матрицей  
Mat_Init_3X2(&ma, 1,0, 0,1, 0,0);
```

### Прототип функции

```
void Mat_Add_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb,  
MATRIX3X3_PTR msum);
```

### Назначение

Функция `void Mat_Add_3X3()` суммирует матрицы ( $ma+mb$ ) и сохраняет результат в матрице `msum`.

### Пример использования

```
MATRIX3X3 m1 = {1,2,3,4,5,6,7,8,9};  
MATRIX3X3 m2 = {4,9,7, -1,5,6, 2,3,4};  
MATRIX3X3 msum;
```

```
// Сложение матриц  
Mat_Add_3X3(&m1, &m2, &msum);
```

### Прототип функции

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb, VECTOR3D_PTR vprod);
```

### Назначение

Функция `void Mat_Mul_VECTOR3D_3X3()` умножает  $1 \times 3$  вектор `va` на матрицу `mb` размером  $3 \times 3$  (рис. 5.18).

### Пример использования

```
VECTOR3D v={x,y,1}, vt;  
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};
```

```
// Умножение  $v \cdot m$   
Mat_Mul_VECTOR3D_3X3(&v, &m, &vt);
```

Можете ли вы сообразить, что именно делает данное преобразование? Если учесть, что `VECTOR3D` на самом деле представляет собой однородные координаты двумерной точки, то становится понятно, что точка перемещается в новое положение, описываемое формулами

```
vt.x = v.x+xt;  
vt.y = v.y+yt;
```

### Прототип функции (из T3DLIB1.CPP|H)

```
int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma, MATRIX3X3_PTR mb,  
MATRIX1X3_PTR mprod);
```

### Назначение

Функция `int Mat_Mul_1X3_3X3()` умножает матрицу  $1 \times 3$  (вектор-строку) на матрицу размером  $3 \times 3$ . Эта функция, за исключением используемых типов, идентична функции `Mat_Mul_VECTOR3D_3X3()`.

### Пример использования

```
MATRIX1X3 v={x,y,1}, vt;  
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};
```

```
// Умножение v*m
Mat_Mul_1X3_3X3(&v, &m, &vt);
```

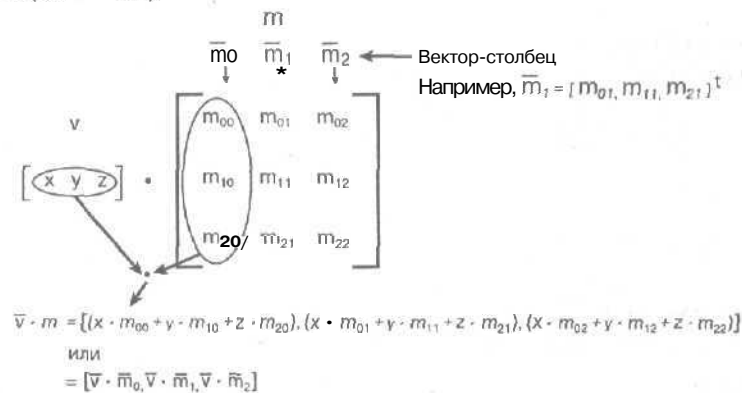


Рис. 5. IS. Умножение трехмерного вектора на матрицу 3x3

#### Прототип функции (из T3DLIB1.CPP|H)

```
int Mat_Mul_3X3(MATRIX3X3_PTR ta, MATRIX3X3_PTR mb,
MATRIX3X3_PTR mprod);
```

#### Назначение

Функция void Mat\_Mul\_3X3() перемножает матрицы размера 3x3 (ma\*mb) и сохраняет результат в матрице mprod.

#### Пример использования

```
MATRIX3X3 m1 = {1,2,3, 4,5,6, 7,8,9};
MATRIX3X3 m2 = {1,0,0, 0,1,0, 0,0,1};
MATRIX3X3 mprod;
```

```
// Умножение. Обратите внимание: m1*m2 = m1, т.к. m2=I
Mat_Mul_3X3(&m1, &m2, &mprod);
```

#### Прототип функции

```
float Mat_Det_3X3(MATRIX3X3_PTR m);
```

#### Назначение

Функция float Mat\_Det\_3X3() вычисляет и возвращает значение определителя матрицы т.

#### Пример использования

```
MATRIX3x3 t = {1,2,0, 4,8,9, 2,5,7};
```

```
// Вычисление определителя
float det = Mat_Det_3X3(&t);
```

#### Прототип функции

```
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
```

#### Назначение

Функция int Mat\_Inverse\_3X3() вычисляет матрицу, обратную матрице т, и сохраняет ее в матрице mi. Если обратная матрица существует, функция возвращает 1; в противном случае функция возвращает 0, а матрица mi является неопределенной.

### Пример использования

```
MATRIX3X3 mA = {1,2,9,4,-3,6, 1,0,5};  
MATRIX3X3 mI;
```

```
// Вычисление обратной к А матрицы  
if (Mat_Inverse_3X3(&mA, &mI))  
{  
    // Обратная матрица существует...  
}  
else  
{  
    // Обратная матрица не существует...  
}
```

### Прототип функции

```
void Print_Mat_3X3(MATRIX3X3_PTR ma, char *name);
```

### Назначение

Функция void Print\_Mat\_3X3() выводит матрицу в удобочитаемом формате вместе с именем, переданным в строке пате. Вывод осуществляется в файл, открытый при помощи вызова функции Open\_Error\_File().

### Пример использования

```
MATRIX3X3 m = {1,2,3, 4,5,6, 7,8,9};  
  
// Открываем файл и выводим в него матрицу  
Open_Error_File("error.txt");  
Print_Mat_3X3(&m, "Matrix m");  
  
// Закрываем файл  
Close_Error_File();
```

### Прототип функции

```
void Mat_Init_4X4(MATRIX4X4_PTR ma,  
    float m00, float m01, float m02, float m03,  
    float m10, float m11, float m12, float m13,  
    float m20, float m21, float m22, float m23,  
    float m30, float m31, float m32, float m33);
```

### Назначение

Функция void Mat\_Init\_4X4() инициализирует матрицу ma непосредственными значениями элементов, передаваемых построчно.

### Пример использования

```
MATRIX4X4 та;  
  
// Инициализация единичной матрицы  
Mat_Init_4X4(&та, 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1);
```

### Прототип функции

```
void Mat_Add_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb,  
    MATRIX4X4_PTR msum);
```

#### Назначение

Функция `void Mat_Add_4X4()` суммирует матрицы ( $ma+mb$ ) и сохраняет результат в матрице `msum`.

#### Пример использования

```
MATRIX4X4 m1 = {1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16};  
MATRIX4X4 m2 = {4,9,7,3, -1,5,6,7, 2,3,4,5, 2,0,5,3};  
MATRIX4X4 msum;
```

```
// Сложение матриц  
Mat_Add_4X4(&m1, &m2, &msum);
```

#### Прототип функции

```
void Mat_Mul_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb,  
                MATRIX4X4_PTR mprod);
```

#### Назначение

Функция `void Mat_Mul_4X4()` перемножает матрицы размером  $4 \times 4$  ( $ma*mb$ ) и сохраняет результат в матрице `mprod`.

#### Пример использования

```
MATRIX4X4 m1 = {1,2,3,4, 4,5,6,7, 7,8,9,10, 11,12,13,14};  
MATRIX4X4 m2 = {1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1};  
MATRIX4X4 mprod;
```

```
// Умножение; обратите внимание:  $m1*m2 = m1$ , т.к.  $m2=I$   
Mat_Mul_4X4(&m1, &m2, &mprod);
```

#### Прототип функции

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma, MATRIX4X4_PTR mb,  
                    MATRIX1X4_PTR mprod);
```

#### Исходный текст функции

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma,  
                    MATRIX4X4_PTR mb,  
                    MATRIX1X4_PTR mprod)  
{  
    // Функция умножает матрицу размером  $1 \times 4$  на  
    // матрицу  $4 \times 4$ . Никаких хитростей, простое  
    // итеративное перемножение  
    for (int col=0; col<4; col++)  
    {  
        // Вычисление скалярного произведения  
        // строки та и столбца mb  
        float sum = 0; // Хранилище результата  
        for (int row=0; row<4; row++)  
        {  
            // Суммирование произведений пар  
            sum += (ma->M[row] * mb->M[row][col]);  
        } // for index  
        // Сохранение полученного элемента  
        mprod->M[col] = sum;  
    } // for col  
} // Mat_Mul_1X4_4X4
```

### Назначение

Функция `void Mat_Mul_1X4_4X4()` умножает матрицу 1x4 на матрицу 4x4 `mprod = (ma*mb)`. Не делается никаких предположений об однородности координат и т.п., выполняется непосредственное умножение матриц. Ускорить функцию можно, используя явное выполнение всех математических операций, без использования циклов.

### Пример использования

```
MATRIX1X4 v={x,y,z,1}, vt;  
MATRIX4X4 m = {1,0,0,0 0,1,0,0 0,0,1,0, xt,yt,zt,1};  
  
// Умножение v*m (перенос трехмерной точки x,y,z)  
Mat_Mul_1X4_4X4(&v, &m, &vt);
```

### Прототип функции

```
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va,  
    MATRIX4X4_PTR mb,  
    VECTOR3D_PTR vprod);
```

### Назначение

Функция `void Mat_Mul_VECTOR3D_4X4()` умножает трехмерный вектор на матрицу 4x4. Для того чтобы такое умножение было осуществимо, функция предполагает наличие фиктивного четвертого элемента вектора, равного 1.0. Кроме того, после выполнения умножения результат представляет собой трехмерный вектор (а не четырехмерный).

Данная функция используется для преобразования трехмерных точек при помощи матриц 4x4 (рис. 5.19).

### Пример использования

```
VECTOR3D v={10,10,10}, vrt;  
  
// Вращение вокруг оси x и перенос (tx,ty,tz)  
MATRIX4X4 m = {1, 0, 0, 0,  
    0, cos(th), sin(th), 0,  
    0, -sin(th), cos(th), 0,  
    tx, ty, tz, 1};  
  
// Выполнение преобразования  
Mat_Mul_VECTOR3D_4X4(&v, &m, &vrt);  
  
Поскольку вектор и точка представляют собой синонимы, те же действия могут быть  
выполнены и с типом POINT3D.  
POINT3D p1={10,10,10}, p2;  
  
// Вращение вокруг оси x и перенос (tx,ty,tz)  
MATRIX4X4 m = {1, 0, 0, 0,  
    0, cos(th), sin(th), 0,  
    0, -sin(th), cos(th), 0,  
    tx, ty, tz, 1};  
  
// Выполнение преобразования  
Mat_Mul_VECTOR3D_4X4(&p1, &m, &p2);
```

### Прототип функции

```
void Mat_Mul_VECTOR3D_4X3(VECTOR3D_PTR va, MATRIX4X3_PTR mb,  
    VECTOR3D_PTR vprod);
```

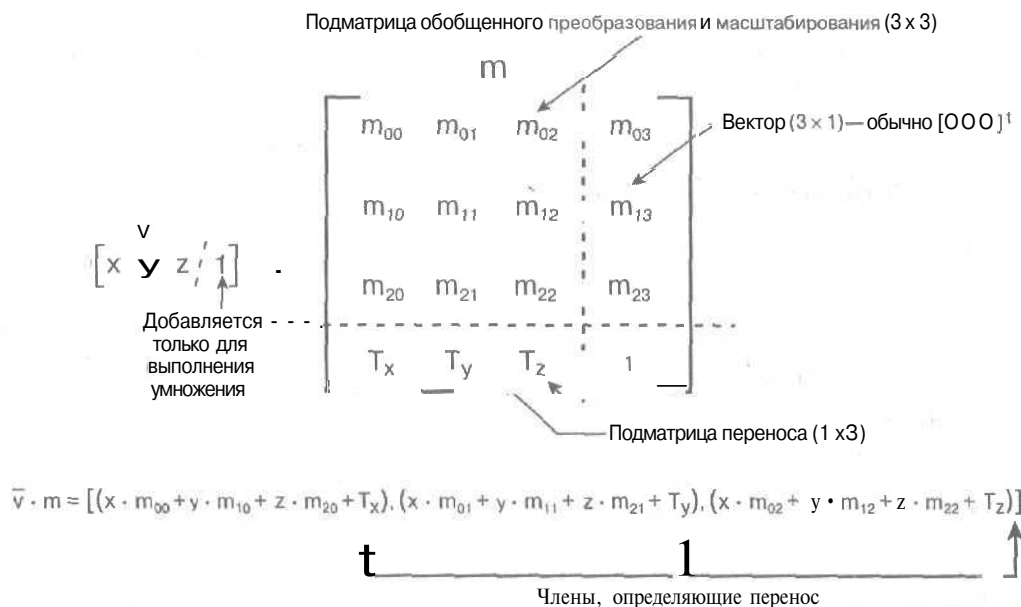


Рис. 5.19. Умножение трехмерного вектора на матрицу 4x4

#### Назначение

Функция `void Mat_Mul_VECTOR3D_4X3()` очень похожа на функцию `Mat_Mul_VECTOR3D_4X4()`, но она выполняет умножение вектора на матрицу размером 4x3, а не 4x4. Здесь также предполагается наличие фиктивного четвертого элемента вектора, равного 1.0. Поскольку в матрице 4x3 имеется только 3 столбца, умножение не требует последующего искусственного отбрасывания элемента, как в случае функции `Mat_Mul_VECTOR3D_4X4()`.

#### Пример использования

```
POINT3D p1={10,10,10}, p2;
// Вращение вокруг оси x и перенос (tx,ty,tz)
// Обратите внимание на отсутствие последнего столбца
MATRIX4X3 m = {1, 0, 0,
                0, cos(th), sin(th),
                0, -sin(th), cos(th),
                tx, ty, tz};
// Выполнение преобразования
Mat_Mul_VECTOR3D_4X3(&p1, &m, &p2);
```

#### Прототип функции

```
void Mat_Mul_VECTOR4D_4X4(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
                           VECTOR4D_PTR vprod);
```

#### Назначение

Функция `void Mat_Mul_VECTOR4D_4X4()` умножает вектор-строку 1x4 `va` на матрицу 4x4 `mb`, сохраняя результат в векторе 1x4 `vprod`. В функции не делается никаких предположений о том или ином виде вектора или матрицы.

#### Пример использования

```
VECTOR4D v={10,10,10,1}, vrt;
// Поворот вокруг оси x и перенос (tx,ty,tz)
```

```

MATRIX4X4 m = { 1, 0, 0, 0,
                 0, cos(th), sin(th), 0,
                 0, -sin(th), cos(th), 0,
                 tx, ty, tz, 1};

```

```

// Выполнение преобразования
Mat_Mul_VECTOR4D_4X4(&v, &m, &vrt);

```

Обратите внимание на то, что результат имеет вид  $vrt = \langle x', y', z', 1 \rangle$ , т.е. компонента  $w$  равна 1.0.

#### Прототип функции

```

void Mat_Mul_VECTOR4D_4X3(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
                           VECTOR4D_PTR vprod);

```

#### Назначение

Функция `void Mat_Mul_VECTOR4D_4X3()` очень похожа на функцию `Mat_Mul_VECTOR3D_4X3()`, но выполняет умножение четырехмерного, а не трехмерного вектора. Поскольку в матрице  $4 \times 3$  три столбца, компонент  $w$  копируется из  $va$  в  $vprod$  (другими словами, дополнительный столбец матрицы  $4 \times 3$  полагается равным  $[0 \ 0 \ 0 \ 1]^T$ ).

#### Пример использования

```

POINT4D p1={10,10,10,1}, p2;

```

```

// Поворот вокруг оси x и перенос (tx,ty,tz)
MATRIX4X3 m = { 1, 0, 0,
                 0, cos(th), sin(th),
                 0, -sin(th), cos(th),
                 tx, ty, tz};

```

```

// Выполнение преобразования
Mat_Mul_VECTOR4D_4X3(&p1, &m, &p2);

```

#### Прототип функции

```

int Mat_Inverse_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mi);

```

#### Назначение

Функция `int Mat_Inverse_4X4()` вычисляет матрицу, обратную матрице  $t$ , и сохраняет ее в матрице  $mi$ . Если обратная матрица существует, функция возвращает 1; в противном случае функция возвращает 0, а матрица  $mi$  является неопределенной. Функция работает только с матрицами, последний столбец которых имеет вид  $[0 \ 0 \ 0 \ 1]^T$ .

#### Пример использования

```

// Обратите внимание на последний столбец
MATRIX4X4 mA = {1, 2, 9, 0,
                 4, -3, 6, 0,
                 1, 0, 5, 0,
                 2, 3, 4, 1};
MATRIX4X4 mI;

```

```

// Вычисляем матрицу, обратную A
if (Mat_Inverse_4X4(&mA, &mI))
    i

```

```

// Обратная матрица существует...
} // if
else
{
// Обратная матрица не существует...
} // else

```

#### Прототип функции

```
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);
```

#### Назначение

Функция void Print\_Mat\_4X4() выводит матрицу в удобочитаемом формате вместе с именем, переданным в строке name. Вывод осуществляется в файл, открытый при помощи вызова функции Open\_Error\_File().

#### Пример использования

```

MATRIX4X4 m = { 1, 2, 3, 4,
                5, 6, 7, 8,
                9, 10, 11, 12,
                13, 14, 15, 16};

```

```
// Открываем файл и осуществляем вывод
```

```

Open_Error_File("error.txt");
Print_Mat_4X4(&m, "Matrix m");

```

```
// Закрываем файл
```

```
Close_Error_File();
```

## Функции для работы с двумерными и трехмерными параметрическими прямыми

Когда я начинал работу над созданием математической библиотеки, я вообще не собирался добавлять в нее поддержку параметрических прямых. Почему? Потому что это все легко кодируется вручную, при помощи вектора  $v$  и параметра  $t$ . Однако затем я подумал, что часто выполняемые вычисления можно закодировать и облегчить тем самым работу программиста. Естественным шагом при этом является решение об инкапсуляции данных, определяющих прямую, в отдельной структуре.

Напомню, как выглядят структуры данных, представляющие параметрические прямые. Вот структура данных для двумерной прямой.

```

// Двумерная параметрическая прямая ///////////////////////////////////
typedef struct PARMLINE2D_TYP
{
    POINT2D p0; // Начальная точка
    POINT2D p1; // Конечная точка
    VECTOR2D v; // Вектор направления
    // |v|=|p0->p1|
} PARMLINE2D, *PARMLINE2D_PTR;

```

Связь значений между собой показана на рис. 5.2. Обратите внимание, что используемый в структуре вектор не нормализован. Структура данных для трехмерной прямой выглядит аналогично.

```

// Трехмерная параметрическая прямая ///////////////////////////////////
typedef struct PARMLINE3D_TYP

```

```

POINT3D p0; // Начальная точка
POINT3D p1; // Конечная точка
VECTOR3D v; // Вектор направления
// |v|=|p0->p1|
} PARMLINE3D, *PARMLINE3D_PTR;

```

Единственным отличием от двумерного варианта является наличие оси z. А теперь можно приступить к рассмотрению функций, работающих с этими структурами.

#### Прототип функции

```

void Init_Parm_Line2D(POINT2D_PTR p_init,
    POINT2D_PTR p_term,
    PARMLINE2D_PTR p);

```

#### Назначение

Функция void Init\_Parm\_Line2D() инициализирует двумерную параметрическую прямую двумя точками и вычисляет вектор между ними.

#### Пример использования

```

POINT2D p1 = {1,2}, p2 = {10,20};
PARMLINE2D p;

```

```

// Создание параметрической прямой от точки p1 к p2
Init_Parm_Line2D(&p1, &p2, &p);

```

#### Прототип функции

```

void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t
    POINT2D_PTR pt);

```

#### Назначение

Функция void Compute\_Parm\_Line2D() вычисляет точку на прямой, соответствующей значению параметра t, и сохраняет его в переменной pt. При t=0 возвращается начальная точка, при t=1 — конечная.

#### Пример использования

```

POINT2D p1 = {1,2}, p2 = {10,20}, pt;
PARMLINE2D p;

```

```

// Создание параметрической прямой от точки p1 к p2
Init_Parm_Line2D(&p1, &p2, &p);

```

```

// Вычисление точки, соответствующей значению t=0.5
Compute_Parm_Line2D(&p, 0.5, &pt);

```

#### Прототип функции

```

int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1,
    PARMLINE2D_PTR p2,
    float *t1, float *t2);

```

#### Исходный текст функции

```

int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1,
    PARMLINE2D_PTR p2,
    float *t1, float *t2)
{
    // Эта функция вычисляет пересечение отрезков двух

```

```

// параметрических прямых и возвращает true, если
// прямые пересекаются. При этом значения t1 и t2
// соответствуют точке пересечения. Параметры могут
// выходить за пределы диапазона [0,1], что означает,
// что несмотря на пересечение прямых отрезки не
// пересекаются. Возвращаемое значение 0 означает,
// что прямые не пересекаются, 1 - что отрезки
// пересекаются, 2 - что прямые пересекаются, но вне
// отрезков, и 3 - что прямые совпадают.

// Шаг 1: проверка на параллельность прямых
float det_plp2 = (p1->v.x*p2->v.y - p1->v.y*p2->v.x);
if (fabs(det_plp2) <= EPSILON_E5)
{
    // Линии либо не пересекаются, либо совпадают.
    // В настоящий момент мы не рассматриваем
    // совпадающие прямые — позже мы можем дописать
    // данную функцию, добавив в нее распознавание
    // совпадения прямых
    return(PARM_LINE_NO_INTERSECT);
} // if
// Шаг 2: Вычисление значений t1 и t2
*t1 = (p2->v.x*(p1->p0.y - p2->p0.y) -
        p2->v.y*(p1->p0.x - p2->p0.x)) / det_plp2;
*t2 = (p1->v.x*(p1->p0.y - p2->p0.y) -
        p1->v.y*(p1->p0.x - p2->p0.x)) / det_plp2;
// Проверка пересечения отрезков
if ((*t1 >= 0) && (*t1 <= 1) && (*t2 >= 0) && (*t2 <= 1))
    return(PARM_LINE_INTERSECT_IN_SEGMENT);
else
    return(PARM_LINE_INTERSECT_OUT_SEGMENT);
} // Intersect_Parm_Lines2D

```

#### Назначение

Функция `int Intersect_Parm_Lines2D()` вычисляет точку пересечения двух параметрических прямых `p1` и `p2`, и возвращает значения параметров точки пересечения `t1` и `t2`. Функция может возвращать следующие значения,

```

// Пересечения нет
#define PARM_LINE_NO_INTERSECT 0
// Пересечение отрезков
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
// Пересечение прямых, но не отрезков
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2

```

В настоящее время функция не рассматривает ситуацию совпадающих прямых из-за множества возможных при этом ситуаций: частичное перекрытие, включение одного отрезка в другой и т.п.). Вы можете добавить необходимую вам функциональность самостоятельно. См. также демонстрационную программу `DEMOIIS_1.CPP|EXE` на прилагаемом компакт-диске.

#### Пример использования

```

// Данные линии пересекаются
POINT2D p1 = {1,1}, p2 = {9,8};

```

```
POINT2D p3 = {2,8}, p4 = {7,1};
PARMLINE2D pl1, pl2;

// Создание параметрических прямых
Init_Parm_Line2D(&p1, &p2, &pl1);
Init_Parm_Line2D(&p3, &p4, &pl2);
float t1=0, t2=0; // Переменные для хранения параметров
// Вычисление точки пересечения
int intersection_type =
    Intersect_Parm_Lines2D(&pl1, &pl2, &t1, &t2);
```

### Прототип функции

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1,
    PARMLINE2D_PTR p2,
    POINT2D_PTR pt);
```

### Назначение

Функция `int Intersect_Parm_Lines2D()` вычисляет точку пересечения двух параметрических прямых, но вместо возврата значений параметров `t1` и `t2` она возвращает координаты точки пересечения. Ясно, что перед тем, как использовать полученную точку, следует проверить, какое **целое** значение вернула данная функция (возвращаемые ею значения такие же, как и у предыдущей функции).

### Пример использования

```
// Эти линии пересекаются
POINT2D p1 = {1,1}, p2 = {9,8};
POINT2D p3 = {2,8}, p4 = {7,1};

POINT2D pt; // Переменная для хранения точки пересечения
PARMLINE2D pl1, pl2;

// Создание параметрических прямых
Init_Parm_Line2D(&p1, &p2, &pl1);
Init_Parm_Line2D(&p3, &p4, &pl2);

// Вычисление точки пересечения
int intersection_type =
    Intersect_Parm_Lines2D(&pl1, &pl2, &pt);
```

**C++**

Программисты на чистом С могут удивиться наличию в одной библиотеке двух функций с одинаковым именем, однако это вполне нормально для С++. Дело в том, что параметры функций различны и, таким образом, с точки зрения компилятора эти функции различны. Такое свойство языка С++ называется *перегрузкой функций* и сопровождается изменением внутреннего представления имени функции с учетом типов ее параметров, что в результате позволяет иметь функции с одинаковыми именами, но разными параметрами.

Теперь перейдем к функциям, предназначенным для работы с трехмерными параметрическими прямыми. В данном разделе их немного, поскольку часть их перенесена в раздел, посвященный функциям для работы с трехмерными плоскостями.

### Прототип функции

```
void Init_Parm_Line3D(POINT3D_PTR p_init,
    POINT3D_PTR p_term,
    PARMLINE3D_PTR p);
```

### Назначение

Функция `void Init_Parm_Line3D()` инициализирует трехмерную параметрическую прямую двумя точками и вычисляет вектор между ними.

### Пример использования

```
POINT3D p1 - {1,2,3}, p2 - {10,20,30};  
PARMLINE3D p;
```

```
// Создание параметрической прямой от точки p1 к p2  
Init_Parm_Line3D(&p1,&p2,&p);
```

### Прототип функции

```
void Compute_Parm_Line3D(PARMLINE3D_PTR p,  
    float t, POINT3D_PTR pt);
```

### Назначение

Функция `void Compute_Parm_Line3D()` вычисляет точку на прямой, соответствующую параметру `t`, и сохраняет ее значение в переменной `pt`. При `t=0` возвращается начальная точка, при `t=1` — конечная.

### Пример использования

```
POINT3D p1 - {1,2,3}, p2 - {10,20,30}, pt;  
PARMLINE3D p;
```

```
// Создание параметрической прямой от точки p1 к p2  
Init_Parm_Line3D(&p1,&p2,&p);
```

```
//Вычисление точки, соответствующей значению t=0.5  
Compute_Parm_Line3D(&p, 0.5, &pt);
```

## Функции для работы с трехмерными плоскостями

Хотя это может выглядеть несколько опрометчиво, я решил добавить в библиотеку поддержку абстрактных трехмерных плоскостей. В действительности 99% времени мы будем работать с замкнутыми многоугольниками (*лежащими* на плоскости), а поэтому, возможно, эти функции позже потребуются *переписать*. Тем не менее, возможность *определять* плоскости и работать с ними нам не повредит на любой стадии работы.

Я решил использовать показанное на рис. 5.20 представление плоскости с помощью точки и вектора нормали:

$$n_x(x - x_0) + n_y(y - y_0) + n_z(z - z_0) = 0,$$

где  $n = \langle n_x, n_y, n_z \rangle$  и  $p_0 = \langle x_0, y_0, z_0 \rangle$ .

В этом случае мы храним *плоскость* в структуре, содержащей вектор нормали и точку на плоскости; вектор нормали при этом вовсе не обязательно должен быть единичным. Напомню, как выглядит соответствующая структура.

```
// Трехмерная плоскость //////////////////////////////////////  
typedef struct PLANE3D_TYP  
{  
    POINT3D pO; // Точка на плоскости  
    VECTOR3D n; // Нормальный (не обязательно  
                // единичный) вектор  
} PLANE3D, *PLANE3D_PTR;
```

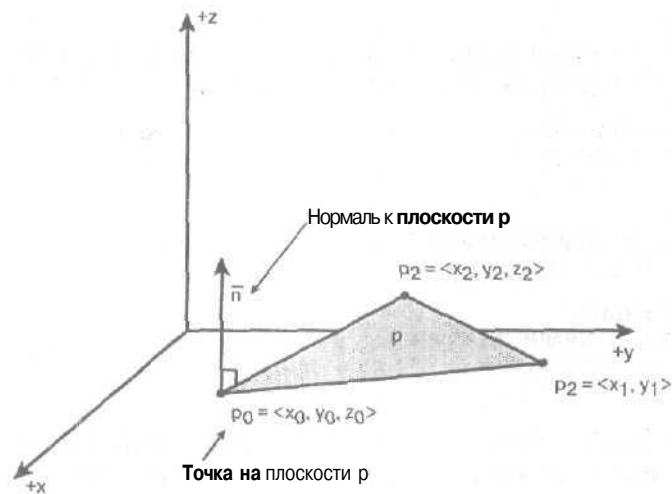


Рис. 5.20. Плоскость, определяемая точкой и нормалью

Многие представленные далее функции тривиальны и призваны всего лишь упростить часто повторяющиеся вычисления либо облегчить представление математических или геометрических объектов.

#### Прототип функции

```
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
    VECTOR3D_PTR normal, int normalize);
```

#### Назначение

Функция void PLANE3D\_Init() инициализирует плоскость при помощи точки и вектора нормали. Дополнительно функция нормализует последний, делая его длину равной 1.0. Для этого параметр normalize следует установить равным TRUE. Такая нормализация нужна при работе с рядом алгоритмов.

#### Пример использования

```
VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// Создание плоскости
PLANE3D_Init(&plane, &p, &n, TRUE);
```

#### Прототип функции

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt
    PLANE3D_PTR plane);
```

#### Исходный текст функции

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt
    PLANE3D_PTR plane)
{
    // Проверка местоположения точки
    // относительно плоскости
    float hs = plane->n.x*(pt->x - plane->p0.x) +
        plane->n.y*(pt->y - plane->p0.y) +
        plane->n.z*(pt->z - plane->p0.z);
```

```

// Указывает полупространство,
// в котором содержится точка
return (hs);
} // Compute_Point_In_Plane3D

```

#### Назначение

Функция `float Compute_Point_In_Plane3D()` обладает достаточно интересной функциональностью. Она вычисляет полупространство, в котором находится переданная ей в качестве параметра точка. Логика данной функции показана на рис. 5.21. Функция **возвращает** значение 0.0, если точка лежит на плоскости, положительное число, если точка находится в положительном полупространстве, и **отрицательное** — если в **отрицательном**. См. также демонстрационную программу `DEMO115_2.CPP|EXE` на прилагаемом компакт-диске.

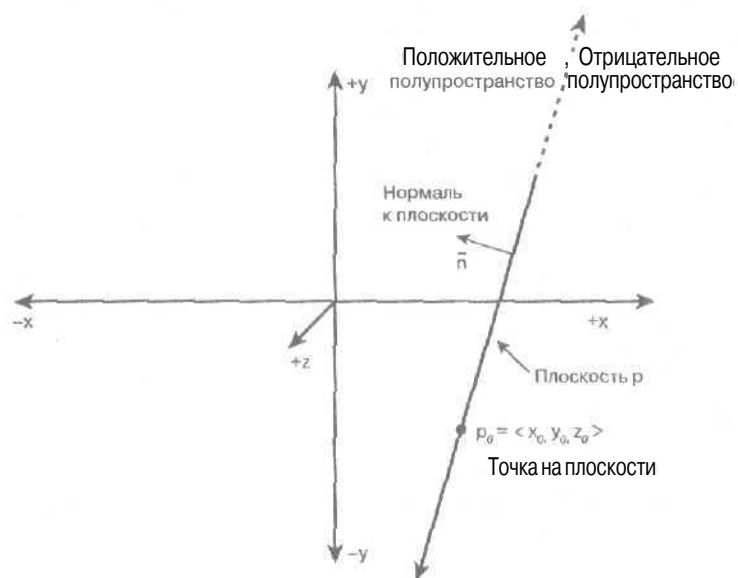


Рис. 5.21. Разбиение пространства плоскостью на два полупространства

#### Пример использования

```

VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// Создание плоскости
PLANE3D_Init(&plane, &p, &n, TRUE);

// Эта точка из положительного полупространства
POINT3D p_test = {50,50,50};

// Проверка местоположения точки
float hs = Compute_Point_In_Plane3D(&p_test, &plane);

```

#### Прототип функции

```

int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline,
    PLANE3D_PTR plane,
    float* t, POINT3D_PTR pt);

```

#### Исходный текст функции

```
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline,
                                   PLANE3D_PTR plane,
                                   float *t, POINT3D_PTR pt)
{
    // Функция определяет, где переданная параметрическая
    // прямая пересекает плоскость. Функция продолжает
    // прямую в обе стороны до бесконечности, однако
    // отрезок пересекает плоскость тогда и только тогда,
    // когда параметр t находится в интервале [0,1].
    // Функция возвращает значение 0, если пересечения нет,
    // 1, если имеется точка пересечения отрезка и
    // плоскости, 2, если точка пересечения находится вне
    // отрезка, и 3, если прямая лежит на плоскости.

    // Проверка параллельности прямой и плоскости
    float plane_dot_line =
        VECTOR3D_Dot(&pline->v, &plane->n);
    if (fabs(plane_dot_line) <= EPSILON_E5)
    {
        // Прямая параллельна плоскости. Не лежит
        // ли она на плоскости?
        if (fabs(Compute_Point_In_Plane3D(
            &pline->p0, plane)
            ) <= EPSILON_E5)
            return(PARM_LINE_INTERSECT_EVERYWHERE);
        else
            return(PARM_LINE_NO_INTERSECT);
    } // if

    // Поиск точки пересечения производится так,
    // как было описано в главе 4. Сначала находим t:
    *t = -(plane->n.x*pline->p0.x +
        plane->n.y*pline->p0.y +
        plane->n.z*pline->p0.z -
        plane->n.x*plane->p0.x -
        plane->n.y*plane->p0.y -
        plane->n.z*plane->p0.z) / (plane_dot_line);
    // Подставляя t в уравнение прямой, находим координаты
    // точки пересечения x,y,z
    pt->x = pline->p0.x + pline->v.x*(*t);
    pt->y = pline->p0.y + pline->v.y*(*t);
    pt->z = pline->p0.z + pline->v.z*(*t);
    // Проверка вхождения t в интервал [0,1]
    if (*t >= 0.0 && *t <= 1.0)
        return(PARM_LINE_INTERSECT_IN_SEGMENT);
    else
        return(PARM_LINE_INTERSECT_OUT_SEGMENT);
} // Intersect_Parm_Line3D_Plane3D
```

#### Назначение

Функция `int Intersect_Parm_Line3D_Plane3D()` вычисляет точку пересечения прямой и плоскости. Функция вычисляет значение параметра `t`, соответствующего пересечению, и точку пересечения `pt`. Однако перед использованием вычисленных значений следует

проверить возвращенное функцией значение, чтобы выяснить, имеет ли место искомое пересечение. Возвращаемые функцией значения те же, что и у функции для вычисления точки пересечения двумерных прямых, с добавлением значения для случая, когда прямая лежит на плоскости.

```
#define PARM_LINE_NO_INTERSECT 0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

См. также демонстрационную программу DEMOII5\_3.CPP|EXE на прилагаемом компакт-диске.

#### Пример использования

```
POINT3D p1 - {5,5,-5}, p2 - {5,5,5},pt;
PARMLINE3D pl;
float t;

// Создание параметрической прямой от точки p1 к p2
// Данная прямая параллельна оси z
Init_Parm_Line3D(&p1,&p2,&pl);
VECTOR3D n={0,0,1};
POINT3D p={0,0,0};
PLANE3D plane;

// Создание плоскости xy
PLANE3D_Init(&plane,&p,&n,TRUE);

// Вычисление точки пересечения
// (которая должна иметь координаты (5,5,0))
int intersection_type =
    Intersect_Parm_Line3D_Plane3D(&pl,&plane,&t,&pt);
```

## Функции для работы с кватернионами

Я уже говорил, что написание функций для работы с кватернионами была не самой простой задачей. Самым сложным было убедиться, что все работает так, как надо. При работе с кватернионами, у которых нет непосредственного физического смысла, оказывается достаточно сложно изобразить происходящее на бумаге и понять, все ли просчитывается, как надо, или в код вкрались ошибки. Но я надеюсь, что я все же смог успешно преодолеть все трудности, и функции для работы с кватернионами работают эффективно и правильно. Как бы то ни было, я действительно люблю кватернионы, которые так упрощают множество сложных операций. Например, поворот вокруг произвольной оси оказывается тривиальной задачей при использовании кватернионов.

Напомню, что кватернионы могут быть записаны одним из следующих способов:

$q = q_0 + q_1i + q_2j + q_3k$ , или

$q = q_0 + \langle q_1, q_2, q_3 \rangle$ , или

$q = q_0 + q_v$ ,

и что тип данных, предназначенный для хранения кватернионов, выглядит следующим образом.

```

// Кватернион //////////////////////////////////////
// Обратите внимание на разные способы представления
// кватерниона с помощью объединения
typedef struct QUAT_TYP
{
    union
    {
        float M[4]; // Массив для хранения данных
        struct
        {
            float q0; // Действительная часть
            VECTOR3D qv; // Векторная часть  $x\mathbf{i}+y\mathbf{j}+z\mathbf{k}$ 
        };
        struct
        {
            float w,x,y,z;
        };
    }; // union
} QUAT, *QUAT_PTR;

```

Эта структура позволяет обращаться к кватерниону как к массиву, действительному числу и вектору или к отдельным его компонентам. А теперь рассмотрим конкретные функции для работы с кватернионами.

#### Прототипы функций

```

void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v,
                             float theta);
void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v,
                             float theta);

```

#### Исходный текст функции

```

void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta)
{
    // Инициализация кватерниона при помощи трехмерного
    // вектора направления и угла. Вектор направления
    // должен быть единичной длины, а угол — выраженным
    // в радианах
    float theta_div_2 = (0.5)*theta; // Находим theta/2
    // Вычисляем кватернион
    float sinf_theta = sinf(theta_div_2);
    q->x = sinf_theta * v->x;
    q->y = sinf_theta * v->y;
    q->z = sinf_theta * v->z;
    q->w = cosf(theta_div_2);
} // VECTOR3D_Theta_To_QUAT

```

#### Назначение

Функция `void VECTOR4D_Theta_To_QUAT()` создает кватернион поворота на основании вектора направления `v` и угла поворота `theta` (рис. 5.22). Функция предназначена для создания кватернионов для расчета поворота точек. Обратите внимание — вектор направления должен быть единичным. В случае передачи в функцию четырехмерного вектора его компонента `w` игнорируется.



Рис. 5,22. Построение кватерниона поворота

#### Пример использования

```
// Создание вектора поворота, представляющего собой
// диагональ первого октанта
VECTOR3D v={1,1,1};
QUAT q;
```

```
// Нормализация v
VECTOR3D_Normalize(&v);
```

```
float theta = DEG_TO_RAD(100); // 100 градусов
```

```
// Создание кватерниона поворота
VECTOR3D_Theta_To_QUAT(&q, &v, theta);
```

#### Прототип функции

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z,
float theta_y, float theta_x);
```

#### Исходный текст функции

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z,
float theta_y, float theta_x)
{
    // Данная функция инициализирует кватернион,
    // основываясь на порядке zyx умножения углов
    // поворотов, параллельных осям z, y, x
    // соответственно. Заметим, что прочие 11 способов
    // задания поворота дают те же результаты
    float cos_z_2 = 0.5*cosf(theta_z);
    float cos_y_2 = 0.5*cosf(theta_y);
    float cos_x_2 = 0.5*cosf(theta_x);
    float sin_z_2 = 0.5*sinf(theta_z);
    float sin_y_2 = 0.5*sinf(theta_y);
    float sin_x_2 = 0.5*sinf(theta_x);
```

```

// Вычисляем кватернион
q->w = cos_z_2*cos_y_2*cos_x_2 +
    sin_z_2*sin_y_2*sin_x_2;
q->x = cos_z_2*cos_y_2*sin_x_2 -
    sin_z_2*sin_y_2*cos_x_2;
q->y = cos_z_2*sin_y_2*cos_x_2 +
    sin_z_2*cos_y_2*sin_x_2;
q->z = sin_z_2*cos_y_2*cos_x_2 -
    cos_z_2*sin_y_2*sin_x_2;
} // EulerZYX_To_QUAT

```

#### Назначение

Функция void EulerZYX\_To\_QUAT() создает кватернион поворота на основании переданных ей углов поворотов Эйлера параллельно осям z, y и x. Эта функция преобразует поворот Эйлера асоответствующий кватернион.

#### Пример использования

```

QUAT qzyx;

// Углы поворота
float theta_x = DEG_TO_RAD(20);
float theta_y = DEG_TO_RAD(30);
float theta_z = DEG_TO_RAD(45);

// Создание кватерниона поворота
EulerZYX_To_QUAT(&qzyx, theta_z, theta_y, theta_x);

```

#### Прототип функции

```

void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v,
    float *theta);

```

#### Исходный текст функции

```

void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v,
    float *theta)
{
    // Данная функция преобразует единичный
    // кватернион в единичный вектор направления
    // и угол поворота вокруг него

    // Получение угла поворота
    *theta = acosf(q->w);

    // Предвычисление для повышения эффективности
    float sinf_theta_inv = 1.0/sinf(*theta);

    // Вычисление вектора
    v->x = q->x*sinf_theta_inv;
    v->y = q->y*sinf_theta_inv;
    v->z = q->z*sinf_theta_inv;

    // Умножение на 2
    *theta*=2;
} // QUAT_To_VECTOR3D_Theta

```

#### Назначение

Функция `void QUAT_To_VECTOR3D_Theta()` преобразует единичный кватернион в единичный вектор направления и угол поворота вокруг него. Эта функция по сути является обратной функции `VECTOR*D_Theta_To_QUAT()`.

#### Пример использования

```
QUAT q;  
// Считаем, что кватернион q представляет собой  
// единичный кватернион поворота  
float theta;  
VECTOR3D v;
```

```
// Преобразуем кватернион в вектор и угол  
QUAT_To_VECTOR3D_Theta(&q, &v, &theta);
```

#### Прототип функции

```
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
```

#### Назначение

Функция `void QUAT_Add()` суммирует кватернионы `q1` и `q2` и сохраняет сумму в переменной `qsum`.

#### Пример использования

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qsum;
```

```
// Сложение кватернионов  
QUAT_Add(&q1, &q2, &qsum);
```

#### Прототип функции

```
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
```

#### Назначение

Функция `void QUAT_Sub()` вычитает кватернион `q2` из `q1` и сохраняет разность в переменной `qdiff`.

#### Пример использования

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qdiff;
```

```
// Вычитание кватернионов  
QUAT_Sub(&q1, &q2, &qdiff);
```

#### Прототип функции

```
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
```

#### Назначение

Функция `void QUAT_Conjugate()` находит кватернион, сопряженный кватерниону `q`, и сохраняет его в переменной `qconj`.

#### Пример использования

```
QUAT q = {1,2,3,4}, qconj;
```

```
// Вычисляем сопряженный кватернион  
QUAT_Conjugate(&q, &qconj);
```

#### Прототип функции

```
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
```

#### Назначение

Функция `void QUAT_Scale()` масштабирует кватернион `q` с коэффициентом `scale` и сохраняет результат в переменной `qs`.

#### Пример использования

```
QUAT q = {1,2,3,4}, qs;
```

```
// Масштабирование q с коэффициентом 2
QUAT_Scale(&q, 2, &qs);
```

#### Прототип функции

```
void QUAT_Scale(QUAT_PTR q, float scale);
```

#### Назначение

Функция `void QUAT_Scale()` масштабирует кватернион `q` с коэффициентом `scale`, непосредственно изменяя при этом значение переменной `q`.

#### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Масштабирование q с коэффициентом 2
QUAT_Scale(&q, 2);
```

#### Прототип функции

```
float QUAT_Norm(QUAT_PTR q);
```

#### Назначение

Функция `float QUAT_Norm(QUAT_PTR q)` возвращает норму (длину) кватерниона `q`.

#### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Чему равна длина q?
float qnorm = QUAT_Norm(&q);
```

#### Прототип функции

```
float QUAT_Norm2(QUAT_PTR q);
```

#### Назначение

Функция `float QUAT_Norm2(QUAT_PTR q)` возвращает квадрат нормы кватерниона `q`. Эта функция очень полезна там, где мы можем использовать **квадрат** нормы — например, при сравнении норм. Преимущество данной функции в более быстром по сравнению с функцией `QUAT_Norm()` вычислении за счет отсутствия вызова `sqrt()`.

#### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Чему равен квадрат длины q?
float qnorm2 = QUAT_Norm2(&q);
```

#### Прототип функции

```
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
```

#### Назначение

Функция `void QUAT_Normalize()` нормализует кватернион `q` и помещает нормализованный кватернион в переменную `qn`.

### Пример использования

```
QUAT q = {1,2,3,4}, qn;
```

```
// Нормализация q
QUAT_Normalize(&q, &qn);
```

### Прототип функции

```
void QUAT_Normalize(QUAT_PTR q);
```

### Назначение

Функция `void QUAT_Normalize()` нормализует кватернион `q`, модифицируя саму переменную `q`.

### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Нормализация q
QUAT_Normalize(&q);
```

### Прототип функции

```
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

### Назначение

Функция `void QUAT_Unit_Inverse()` вычисляет кватернион, обратный кватерниону `q`, и сохраняет его в переменной `qi`. Кватернион `q` должен быть единичным, поскольку функция использует тот факт, что обратным к единичному кватерниону является сопряженный кватернион.

### Пример использования

```
QUAT q = {1,2,3,4}, qi;
```

```
// Сначала нормализуем q
QUAT_Normalize(&q);
```

```
// Вычисляем обратный кватернион
QUAT_Unit_Inverse(&q, &qi);
```

### Прототип функции

```
void QUAT_Unit_Inverse(QUAT_PTR q);
```

### Назначение

Функция `void QUAT_Unit_Inverse()` вычисляет кватернион, обратный кватерниону `q`, сохраняя его в той же переменной `q`. Кватернион должен быть единичным, поскольку функция использует тот факт, что обратным единичному кватерниону является сопряженный кватернион.

### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Сначала нормализуем q
QUAT_Normalize(&q);
```

```
// Вычисляем обратный кватернион
QUAT_Unit_Inverse(&q);
```

#### Прототип функции

```
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

#### Назначение

Функция void `QUAT_Inverse()` вычисляет кватернион, обратный произвольному (требование единичности отсутствует) кватерниону `q`, и сохраняет его в переменной `qi`.

#### Пример использования

```
QUAT q = {1,2,3,4}, qi;
```

```
// Вычисляем обратный кватернион
QUAT_Inverse(&q, &qi);
```

#### Прототип функции

```
void QUAT_Inverse(QUAT_PTR q);
```

#### Назначение

Функция void `QUAT_Inverse()` вычисляет кватернион, обратный произвольному (требование единичности отсутствует) кватерниону `q`, и сохраняет его в той же переменной `q`.

#### Пример использования

```
QUAT q = {1,2,3,4};
```

```
// Вычисляем обратный кватернион
QUAT_Inverse(&q);
```

#### Прототип функции

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
```

#### Исходный текст функции

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod)
```

```
{
    // Функция перемножает два кватерниона методом "в лоб"
    // qprod->w = q1->w*q2->w - q1->x*q2->x -
    //          q1->y*q2->y - q1->z*q2->z;
    // qprod->x = q1->w*q2->x + q1->x*q2->w +
    //          q1->y*q2->z - q1->z*q2->y;
    // qprod->y = q1->w*q2->y - q1->x*q2->z +
    //          q1->y*q2->w - q1->z*q2->x;
    // qprod->z = q1->w*q2->z + q1->x*q2->y -
    //          q1->y*q2->x + q1->z*q2->w;

    // Для уменьшения количества умножений
    // используется выделение множителей
    float prd_0 = (q1->z - q1->y) * (q2->y - q2->z);
    float prd_1 = (q1->w + q1->x) * (q2->w + q2->x);
    float prd_2 = (q1->w - q1->x) * (q2->y + q2->z);
    float prd_3 = (q1->y + q1->z) * (q2->w - q2->x);
    float prd_4 = (q1->z - q1->x) * (q2->x - q2->y);
    float prd_5 = (q1->z + q1->x) * (q2->x + q2->y);
    float prd_6 = (q1->w + q1->y) * (q2->w - q2->z);
    float prd_7 = (q1->w - q1->y) * (q2->w + q2->z);
    float prd_8 = prd_5 + prd_6 + prd_7;
    float prd_9 = 0.5 * (prd_4 + prd_8);
    // Сборка результата из временных переменных

```

```

qprod->w = prd_0 + prd_9 - prd_5;
qprod->x = prd_1 + prd_9 - prd_8;
qprod->y = prd_2 + prd_9 - prd_7;
qprod->z = prd_3 + prd_9 - prd_6;
} // QUAT_Mul

```

Использование непосредственного определения произведения кватернионов приводит к 16 умножениям и 12 сложениям. Выполнив несложные алгебраические преобразования, я получил 9 умножений и 27 сложений. Проблема только в том, что при использовании сопроцессора для работы с числами с плавающей точкой преимущество может быть **слишком незначительным** или отсутствовать вовсе.

#### Назначение

Функция void QUAT\_Mul() перемножает кватернионы  $q_1 * q_2$  и сохраняет результат в переменной qprod.

#### Пример использования

QUAT q1={1,2,3,4}, q2={5,6,7,8}, qprod;

```

// Умножение q1*q2
QUAT_Mul(&q1, &q2, qprod);

```

#### ВНИМАНИЕ

Не забывайте, что произведение кватернионов в общем случае **некоммутативно**, т.е.  
 $q_1 q_2 \neq q_2 q_1$ .

#### Прототип функции

```

void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2,
    QUAT_PTR q3, QUAT_PTR qprod);

```

#### Назначение

Функция void QUAT\_Triple\_Product() перемножает три кватерниона  $q_1 * q_2 * q_3$  и сохраняет результат в переменной qprod. Эта функция полезна при повороте точки, т.к. поворот требует перемножения трех кватернионов:  $q * vq$  и  $qvq^*$ .

#### Пример использования

```

// Поворот точки {5,0,0} вокруг оси z на 45 градусов

// Шаг 1: создание кватерниона поворота
VECTOR3D vz = {0,0,1};
QUAT qr, // Здесь будет храниться кватернион поворота
qrc; // и сопряженный к нему кватернион

// Создание кватерниона поворота
VECTOR3D_Theta_To_QUAT(&qr, &vz, DEG_TO_RAD(45));

// Вычисление сопряженного кватерниона
QUAT_Conjugate(&qr, &qrc);

// Создаем точку, которая будет поворачиваться
// (q0 = 0; x,y,z - координаты точки)
QUAT qp={0,5,0,0};

// Выполняем поворот точки p вокруг оси z на 45 градусов
QUAT_Triple_Product(&qr, &qp, &qrc, &qprod);

```

```
// В полученном кватернионе qO=0. Для получения
// координат точки после поворота следует взять
// компоненты x,y и z
```

#### Прототип функции

```
void QUAT_Print(QUAT_PTR q, char *name);
```

#### Назначение

Функция `void QUAT_Print()` выводит кватернион в удобочитаемом формате вместе с именем, переданным в строке `name`. Вывод осуществляется в файл, открытый при помощи вызова функции `Open_Error_File()`.

#### Пример использования

```
// Открываем вывод на экран
Open_Error_File("", stdout);
QUAT q={1,2,3,4};
QUAT_Print(&q);

// Закрываем файл
Close_Error_File();
```

На прилагаемом компакт-диске имеется демонстрационная программа для работы с кватернионами — `DEMO115_4.CPP|EXE`. Она позволяет вам ввести два кватерниона и точку в трехмерном пространстве и выполнить над ними различные операции, описанные в этом разделе.

## Функции для работы с числами с фиксированной точкой

Хотя тема работы с числами с фиксированной точкой раскрыта в предыдущей главе далеко не полно, я, тем не менее, оснастил математическую библиотеку рядом соответствующих функций. Если вы хотите познакомиться с числами с фиксированной точкой более подробно — приобретите мою предыдущую книгу<sup>1</sup>, там этот материал изложен более подробно. Кроме того, современные процессоры работают с числами с плавающей точкой практически так же быстро, как и с целыми числами, поэтому использование математики с фиксированной точкой становится не столь уж необходимым. Тем не менее, имеется ряд алгоритмов, где использование чисел с фиксированной точкой вполне оправдано.

Числа с фиксированной точкой — старый хорошо известный трюк, использовавшийся на медленных процессорах. В нем используется "фиксированная точка", которая искусственно разделяет целое число на целую и дробную части. Я использую формат чисел с фиксированной точкой 16.16, который означает, что у нас имеется 16 бит в целой, и 16 бит в дробной части числа. Таким образом, диапазон представления чисел с фиксированной точкой —  $\pm 32768$ , с точностью  $2^{-16}$ .

При работе с числами с плавающей точкой неизбежно возникает вопрос о том, каким образом они должны быть представлены. Для этих чисел можно использовать обычное 32-битовое целое число (или, если его размер слишком мал — то 64-битовое целое). Соответственно, вот как выглядят определенные в файле `T3DLIB4.H` типы данных для чисел с фиксированной точкой.

```
// Числа с фиксированной точкой //////////////////////////////////
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

<sup>1</sup> Андре Ламот. Программирование игр для Windows. Советы профессионала, 2-е изд. — М.: Издательский дом "Вильямс", 2003. — Прим. ред.

В файле T3DLIB1.H определены несколько констант, связанных с числами с фиксированной точкой и призванных упростить работу программиста.

```
// Константы, связанные с числами с фиксированной точкой
#define FIXP16_SHIFT 16
#define FIXP16_MAG 65536
#define FIXP16_DP_MASK 0x0000ffff
#define FIXP16_WP_MASK 0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

### Создание числа с фиксированной точкой из целого

Для создания числа с фиксированной точкой из целого числа следует выполнить сдвиг последнего на `FIXP16_SHIFT` бит влево, что приводит к размещению исходного числа в целой части числа с фиксированной точкой. Важно только не забывать о возможности переполнения последнего, т.к. для целой части в нем отводится в 2 раза меньше битов, чем в целом числе. Вот как выполняется описанное действие:

```
FIXP16 fp1 = (100 << FIXP16);
```

Те же действия выполняет макрос `INT_TO_FIXP16()`, входящий в состав математической библиотеки.

### Создание числа с фиксированной точкой из числа с плавающей точкой

Здесь ситуация немного сложнее, поскольку бинарные представления этих чисел совершенно различны. Поэтому вместо сдвига в данном случае следует использовать умножение на  $2^{16} = 65536$ :

```
FIXP16 fp1 = (int)(100.5*65536.0);
```

Те же действия выполняет макрос `FLOAT_TO_FIXP16()`, входящий в состав математической библиотеки. В данной ситуации вы можете также захотеть округлить получаемое число, так как в процессе преобразования типа в `int` происходит отбрасывание дробной части числа. Для этого достаточно перед выполнением преобразования прибавить 0.5:

```
FIXP16 fp1 = (int)(100.5*65536.0 + 0.5);
```

### Обратное преобразование в число с плавающей точкой

Для того чтобы преобразовать число с фиксированной точкой в число с плавающей точкой, достаточно поделить его на 65536:

```
float ((float)fp)/65536.0;
```

(В математической библиотеке это делает макрос `FIXP16_TO_FLOAT()`).

Теперь кратко рассмотрим выполнение некоторых операций над числами с фиксированной точкой.

### Сложение и вычитание

Для сложения или вычитания двух чисел, представленных в виде чисел с фиксированной точкой, достаточно просто сложить их или вычесть, как обычные целые числа.

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);
```

```
FIXP16 fpsum = fp1 + fp2;
FIXP16 fpdiff = fp1 - fp2;
```

А вот более наглядный пример.

```
int x=50, y=23;

// Преобразуем x и y в числа с фиксированной точкой
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
FIXP16 fsum;
```

```
// Складываем их
fsum = fpx + fpy;
```

Эти действия упрощаются до

```
fsum = x*65536 + y*65536 = (x+y)*65536;
```

Как видите, полученный результат действительно представляет собой сумму  $(x + y)$ , умноженную на 65536 — т.е. приведенную к формату с фиксированной точкой.

## Умножение

Выполнить умножение чисел с фиксированной точкой несколько сложнее. Причина в том, что каждое число оказывается умноженным на **масштабирующий** множитель 65536. Таким образом, при непосредственном умножении двух чисел результат оказывается умноженным на  $65536^2$ .

```
// Числа, преобразуемые в формат с фиксированной точкой
int x=50, y=23;
```

```
// Преобразование в формат 16.16
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

Попробуем перемножить полученные числа.

```
fpx*fpy - (x*65536) * (y*65536) - (x*y)*(65536*65536);
```

Теперь вы видите, в чем состоит проблема? Полученный результат оказывается масштабированным с коэффициентом  $65536^2$ , а не с **требуемым** 65536. Решение простое: надо выполнить обратное масштабирование. Но главная проблема не в этом: перемножение двух чисел с фиксированной точкой вызывает переполнение 32-битового представления. Поэтому реальными решениями являются либо использование 64-битовой математики, либо предварительное масштабирование сомножителей так, чтобы произведение было масштабировано с корректным коэффициентом 65536. Вот как это можно сделать:

```
= (fpx/256)*(fpy/256)
```

Это приводит к следующему результату:

```
= ((x*65536)/256) * ((y*65536)/256)
= (x*256)*(y*256) = (x*y)*65536
```

Казалось бы, все в порядке, но данный способ имеет один серьезный недостаток — в процессе деления на 256 мы теряем 8 бит точности как у множимого, так и у **множителя**, что неприемлемо: ведь главная цель математики с фиксированной точкой состоит в обеспечении точности при работе с числами с десятичной точкой. В этой ситуации может выручить ассемблерная вставка с использованием 64-битовой математики.

## Деление

Деление сопряжено с проблемой, обратной умножению: при делении чисел мы теряем десятичную точку, т.е. полученный результат оказывается масштабированным с коэффициентом 1, а не 65536.

```
// Числа, преобразуемые в формат с фиксированной точкой
int x=50, y=23;
```

```
// Преобразование в формат 16.16
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

Теперь выполним деление и посмотрим на результат:

```
fpx/fpy - (x*65536) / (y*65536) - (x/y);
```

Определенно, это не то, что нам надо. Мы вновь оказываемся перед дилеммой — использовать ассемблер для осуществления 64-битового деления или масштабировать числитель и знаменатель таким образом, чтобы в итоге получить корректно масштабированное частное? В последнем случае мы опять сталкиваемся с проблемой потери точности. Для того чтобы результат деления был корректно масштабирован (с коэффициентом 65536), можно увеличить числитель в 256 раз, а знаменатель в 256 раз уменьшить. Главным недостатком этого способа является то, что по сути при этом мы делим число в формате 8.16 на число в формате 16.8, так что числитель не может превышать 255, а знаменатель — быть меньше  $1/2^8 = 0.0039$ . Вот как осуществляется описанный способ деления.

```
// Числа, преобразуемые в формат с фиксированной точкой
int x=50, y=23;
```

```
// Преобразование в формат 16.16
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

```
// Деление
fpx/fpy = (x*256*65536) / (y*65536/256)
        - (x*256*256/y)*65536/65536
        - (x/y)*(256*256) - (x/y)*65536
```

**СОВЕТ**

Как видите, числа с фиксированной точкой не обязаны быть только в одном формате 16.16 — одни числа в нашей программе могут быть в этом формате, другие — в формате 24.8, а третьи и вовсе 0.32. Главное — корректное масштабирование и отслеживание возможных переполнений и потерь точности при выполнении математических операций.

Теперь можно приступить к рассмотрению конкретных функций математической библиотеки.

### Прототип функции

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
```

### Исходный текст функции

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2)
{
    // Функция вычисляет произведение fp_prod = fp1*fp2
    // с использованием 64-битной математики
    FIXP16 fp_prod; // Возвращаемое значение
```

```

_asm {
    mov eax, fp1
    imul fp2    // Умножение fp1*fp2
    shrd eax, edx, 16 // Результат в формате 32:32
    // размещен в регистрах edx:eax. Приводим его
    // к виду 16:16 в регистре eax
} // asm
} // FIXP16_MUL

```

Данный код очень прост и понятен. Самое интересное в нем то, что 32-битовые процессоры Intel имеют поддержку 64-битовой математики.

#### Назначение

Функция `FIXP16 FIXP16_MUL()` умножает два числа с фиксированной точкой `fp1*fp2` с использованием 64-битовой математики и возвращает полученное произведение. Заметим, что при этом нет потерь точности.

#### Пример использования

```

FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

```

```

// Выполняем умножение
FIXP16 fpprod = FIXP16_MUL(fp1, fp2);

```

#### Прототип функции

```

FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2);

```

#### Исходный текст функции

```

FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2)
{
    // Функция вычисляет частное fp1/fp2 с использованием
    // 64-битной математики без потери точности
    _asm {
        mov eax, fp1    // Помещаем делимое в eax
        cdq             // Знаковое расширение в edx:eax
        shld edx, eax, 16 // Сдвиг 16:16 в edx
        sal eax, 16     // Сдвиг eax
        idiv fp2        // Выполнение деления
        // Результат находится в eax
    } // asm
} // FIXP16_DIV

```

Алгоритм деления немного сложнее. Сначала делимое должно быть **знаково** расширено до 64 бит и выполнен сдвиг для корректного масштабирования результата. Этот сдвиг выполняется двумя операциями, поскольку команда `shld` не выполняет сдвиг в регистре `eax`. Более полную информацию об этих несуразностях при реализации 64-битовых сдвигов в процессорах Intel можно найти в любом руководстве по их ассемблеру.

#### Назначение

Функция `FIXP16 FIXP16_DIV()` выполняет деление `fp1/fp2` и возвращает полученный результат. В связи с использованием 64-битовой математики потери точности при этом отсутствуют.

#### Пример использования

```

FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

```

```
// Выполняем деление
FIXP16 fpdiv = FIXP16_DIV(fp1,fp2);
```

#### НА ЗАМЕТКУ

Как видите, для выполнения умножения и деления чисел с фиксированной точкой используется несколько команд процессора. Следует заметить, что на современных процессорах сложение и вычитание чисел с фиксированной точкой выполняется с той же скоростью, что и сложение и вычитание чисел с плавающей точкой, а умножение и деление последних по скорости зачастую превосходит соответствующие операции над числами с фиксированной точкой.

#### Прототип функции

```
void FIXP16_Print(FIXP16 fp);
```

#### Назначение

Функция `void FIXP16_Print()` выводит число с фиксированной точкой, как если бы это было число с плавающей точкой.

#### Пример использования

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16_Print(fp1);
```

На прилагаемом компакт-диске имеется демонстрационная программа для работы с числами с фиксированной точкой — `DEMO115_5.CPP|EXE`. Она позволяет вам ввести два числа с плавающей точкой, преобразует их в числа с фиксированной точкой, выполняет над ними все описанные операции и выводит результаты, чтобы вы могли увидеть точность выполняемых действий.

## Функции для решения систем уравнений

Следующие две функции предназначены для решения систем линейных уравнений вида  $A \cdot X = B$ . Все, что вам надо передать в функцию в качестве параметра — это матрицу коэффициентов  $A$  и матрицу свободных членов  $B$ , а также матрицу, в которую следует поместить решение, если таковое существует. Например, пусть у нас имеется система уравнений

$$3x + 2y - 5z = 6,$$

$$x - 3y + 7z = -4,$$

$$5x + 9y - 2z = 5.$$

В таком случае матрицы  $A$ ,  $B$  и  $X$  выглядят следующим образом:

$$A = \begin{bmatrix} 3 & 2 & -5 \\ 1 & -3 & 7 \\ 5 & 9 & -2 \end{bmatrix}, X = \begin{bmatrix} x & y & z \end{bmatrix}^T \text{ И } B = \begin{bmatrix} 6 & -4 & 5 \end{bmatrix}^T.$$

#### Прототип функции

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X
MATRIX1X2_PTR B);
```

#### Исходный текст функции

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X
MATRIX1X2_PTR B)
{
    // Решает систему уравнений AX=B и вычисляет X=A(-1)*B
```

```

// с использованием правила Крамера

// Шаг 1: вычисление определителя A
float det_A = Mat_Det_2X2(A);
// Проверка на равенство det(a) нулю (если это так,
// решения не существует)
if (fabs(det_A) < EPSILON_E5)
    return (0);
// Шаг 2: Создаем матрицы-числители путем замены
// соответствующих столбцов матрицы A транспонированной
// матрицей B и находим решение системы уравнений
MATRIX2X2 work_mat; // Рабочая матрица
// Поиск x
// Копируем A в рабочую матрицу
MAT_COPY_2X2(A, &work_mat);
// Замена столбца x
MAT_COLUMN_SWAP_2X2(&work_mat, 0, B);
// Вычисление определителя
float det_ABx = Mat_Det_2X2(&work_mat);
// Поиск значения x
X->M00 = det_ABx/det_A;
// Поиск y
// Копируем A в рабочую матрицу
MAT_COPY_2X2(A, &work_mat);
// Замена столбца y
MAT_COLUMN_SWAP_2X2(&work_mat, 1, B);
// Вычисление определителя
float det_ABy = Mat_Det_2X2(&work_mat);
// Поиск значения y
X->M01 = det_ABy/det_A;
// Возврат кода успешного завершения
return(1);
} // Solve_2X2_System

```

#### Назначение

Функция `int Solve_2X2_System()` предназначена для решения системы линейных уравнений  $A \cdot X = B$ , где  $A$  имеет размер  $2 \times 2$ , а матрицы  $B$  и  $X$  — размер  $1 \times 2$ . Если решение существует, оно сохраняется в матрице  $X$  и функция возвращает значение 1; в противном случае функция возвращает значение 0, а матрица  $X$  не определена. Функция для решения системы уравнений использует правило Крамера. Это неплохой пример использования функций для работы с матрицами, так что присмотритесь к нему повнимательнее. На прилагаемом компакт-диске имеется демонстрационная программа `DEMO115_6.CPP|EXE`, которая позволяет вам ввести и решить систему линейных уравнений.

(Следует заметить, что приведенный исходный текст является примером использования функций, предназначенных для работы с матрицами, но никак не примером эффективного решения системы линейных уравнений с двумя неизвестными. Гораздо более эффективное решение без излишних пересылок в памяти может выглядеть следующим образом.

```

int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X,
                    MATRIX1X2_PTR B)
{
    float det_A = Mat_Det_2X2(A);
    if (fabs(det_A) < EPSILON_E5)

```

```

    return (0);
    X->M00 = (B->M00*A->M11 - B->M01*A->M01)/det_A;
    X->M01 = (B->M01*A->M00 - B->M00*A->M10)/det_A;
    return (1);
} // Solve_2X2_System

```

— Прим. ред.)

#### Пример использования

В качестве демонстрационного примера приведен код функции `main()` демонстрационной программы `DEMOII5_6.CPP`.

`void main()`

```

{
    MATRIX2X2 mA;
    MATRIX1X2 mB;
    MATRIX1X2 mX;
    // Для вывода сообщений используем экран
    Open_Error_File("", stdout);
    // Ввод матриц А и В
    printf("\nEnter the values for the matrix A (2x2)");
    printf("\nin row major form m00, m01, m10, m11?");
    scanf("%f, %f, %f, %f", &mA.M00,
        &mA.M01, &mA.M10, &mA.M11);
    printf("\nEnter the values for matrix B (2x1)");
    printf("\nin column major form m00, m10?");
    scanf("%f%f", &mB.M00, &mB.M01);
    // Решаем систему уравнений...
    if (Solve_2X2_System(&mA, &mX, &mB))
    {
        // ...и выводим результат
        VECTOR2D_Print((VECTOR2D_PTR)&mX,
            "Solution matrix mX");
    } // if
    else
        // ...или сообщение об ошибке
        printf("\nNo Solution!");
    // Закрываем файл вывода
    Close_Error_File();
} // main

```

#### Прототип функции

```

int Solve_3X3_System(MATRIX3X3_PTR A, MATRIX1X3_PTR X
    MATRIX1X3_PTR B);

```

#### Назначение

Функция `int Solve_3X3_System()` предназначена для решения системы линейных уравнений  $A \cdot X = B$ , где  $A$  имеет размер  $3 \times 3$ , а матрицы  $B$  и  $X$  — размер  $1 \times 3$ . Если решение существует, оно сохраняется в матрице  $X$  и функция возвращает значение 1; в противном случае функция **возвращает** значение 0, а матрица  $X$  не определена.

#### Пример использования

```

MATRIX3X3 mA = {1,2,9, 4,-3,6, 1,0,5};
MATRIX1X3 mB = {1,2,3};
MATRIX1X3 mX;

```

```
// Решаем систему уравнений
if(Solve_3X3_System(&mA, &mX, &mB))
{
    // Вывод результатов
    VECTOR3D_Print((VECTOR3D_PTR)&mX, "Solution matrix mX");
}
> // if
else
    printf("\nNo Solution!");
```

## Работа математического сопроцессора

Старый сопроцессор 80837 и встроенный математический сопроцессор **PentiumX** (далее просто FPU — floating point unit), **вероятно**, можно считать наиболее загадочными частями компьютера. Откровенно говоря, найти хорошее руководство по программированию FPU очень трудно — их попросту можно пересчитать по пальцам. Я хочу попытаться поправить ситуацию — по крайней мере, для читателей этой книги.

### НА ЗАМЕТКУ

Современные процессоры Pentium могут иметь несколько сопроцессоров, однако я буду говорить об FPU обобщенно, не рассматривая отдельно случай с несколькими процессорами.

Конечно, ваш компилятор **C/C++** отлично **справляется** со своими обязанностями и вполне успешно использует FPU при работе с числами с **плавающей** точкой. Тем не менее, генерируемый компилятором код в критичных участках можно попытаться оптимизировать вручную при помощи ассемблера и 64-битовых инструкций — например, как было сделано при рассмотрении умножения и деления чисел с фиксированной точкой.

Есть и другой аспект работы с **сопроцессором**. В моделях Pentium с поддержкой **MMX** регистры FPU представляют собой синонимы регистров MMX, что означает, что вы не можете работать с кодом MMX и кодом FPU без постоянного переключения контекстов. Работа с **MMX** — **отдельная** тема, которой я не намерен касаться. Если она интересует вас, вы можете обратиться к неплохому руководству Intel *MMX Programming Guide*, которое можно найти в разделе для программистов на Web-узле Intel.

Итак, как же работает FPU? С точки зрения программирования, не имеет значения, один ли сопроцессор в системе или несколько, внешний он или встроенный — набор команд остается практически один и тот же. Именно эти **общие** команды нас и интересуют.

### ВНИМАНИЕ

Программирование FPU должно выполняться на уровне ассемблера; мы будем использовать встраиваемый ассемблер. Если вы еще не стали профессионалом в этой области, не беспокойтесь: программирование FPU — достаточно простая вещь, поскольку сопроцессор очень похож на всего лишь маленький компьютер со стеком.

## Архитектура сопроцессора

Сопроцессор представляет собой виртуальную стековую машину, предназначенную для вычисления математических операций с числами с **плавающей** точкой. На рис. 5.23 показана абстрактная модель FPU и его взаимоотношения с процессором. Все команды, передаваемые процессору и предназначены сопроцессору, передаются для выполнения последнему, никак не затрагивая процессор и его внутренние регистры. Хотя часто процессор и сопроцессор не могут работать одновременно, все же в **большинстве** случаев они в состоянии работать параллельно, в особенности в случае архитектуры с использованием U-V-каналов наподобие Pentium.

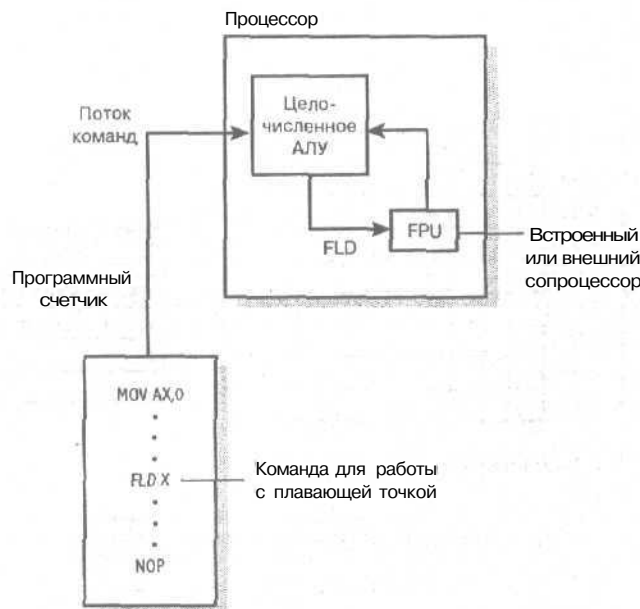


Рис. 5.23. Математические операции с плавающей точкой выполняются сопроцессором

Сопроцессор может выполнять множество математических операций, как стандартных (таких как сложение, вычитание, умножение и деление), так и более сложных (логарифмирование, вычисление различных трансцендентных функций и т.п.). Более того, сопроцессор в состоянии выполнять многие операции быстрее, чем процессор — соответствующие операции с целыми числами, и, кроме того, с более высокой точностью. Даже в случае, когда FPU оказывается более медленным, чем целочисленный процессор, это компенсируется повышенной точностью — что приобретает особое значение в трехмерной графике.

Программирование FPU — очень простая задача. Он имеет свой набор команд, с которыми вы вскоре познакомитесь. Обычно, когда вы хотите выполнить ту или иную команду FPU, вы просто помещаете ее в ваш код при помощи соответствующей ассемблерной команды. Это создает определенную сложность, так как для этого надо либо использовать внешние функции, написанные на ассемблере, либо встраиваемый ассемблер. Лично я предпочитаю последний способ, кодируя небольшие функции полностью на встраиваемом ассемблере (ясно, что я не делаю этого для объемных высокоуровневых функций).

## Стек сопроцессора

Поговорим о внутреннем стеке сопроцессора, поскольку именно на нем строится все работа. Обычно в стеке FPU имеется восемь элементов и слово состояния (рис. 5.24). Каждый элемент стека имеет размер 80 битов, из которых 64 бита отведено под десятичное значение, 14 — для показателя степени и один бит — знаковый.

Однако каждый элемент стека может хранить данные разных типов, как показано на рис. 5.25. Как видите, это могут быть 4-, 8- и 10-байтовые значения с плавающей точкой. Элементы стека используются для передачи входных данных и получения результата вычислений, и в этом смысле они являются регистрами сопроцессора, т.е. их можно рассматривать и как набор регистров, и как стек.

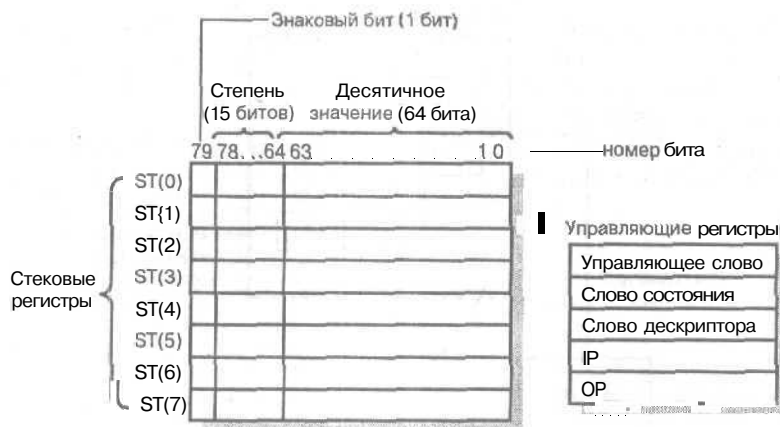


Рис. 5.24. Базовая архитектура FPU

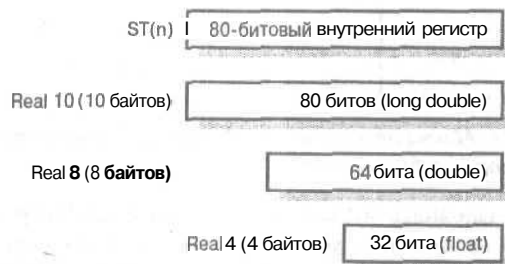


Рис. 5.25. Форматы данных FPU<sup>2</sup>

Обращение к элементам стека осуществляется с использованием синтаксиса  $ST(n)$ , где  $n$  — номер элемента. Рассмотрим рис. 5.26.



Рис. 5.26. Обращение к элементам стека сопроцессора

<sup>2</sup> Заметим, что стандарт C/C++ не оговаривает размер чисел float, double и long double, так что точного соответствия размера типу, показанного на рисунке, у конкретного компилятора может и не быть. В качестве примера можно привести Watcom C++, у которого `sizeof(double) = sizeof(long double)`. — Прим. ред.

ST эквивалентно вершине стека (TOS, top of stack).

ST(0) эквивалентно вершине стека.

ST(1) эквивалентно второму элементу стека.

ST(7) эквивалентно последнему элементу стека.

При программировании FPU многие команды принимают ноль, один или два операнда. Операндами в большинстве случаев являются константы, память или элементы стека. Большинство команд помещают результат вычислений на вершину стека, в ST(0). При работе помогает также знание о том, какие данные после выполнения операции остаются доступны и корректны, а какие уничтожаются.

Теперь, когда вы получили общее представление о сопроцессоре, перейдем к набору его команд.

**НА ЗАМЕТКУ**

Поскольку выпускаются все новые и новые процессоры Pentium, пытаться составить полный список команд сопроцессора — безнадежная задача. В табл. 5.1 перечислены основные команды сопроцессора, из которых вы, вероятно, будете использовать в лучшем случае десятую часть.

## Набор команд сопроцессора

FPU представляет собой полноценный процессор с большим набором команд, которые могут быть разделены на несколько категорий:

- передача данных между процессором и сопроцессором;
- арифметические операции;
- трансцендентные функции, такие как синус, косинус и т.п.;
- константы наподобие числа  $\pi$ ;
- операции сравнения чисел с плавающей точкой;
- управляющие команды.

В табл. 5.1 перечислены команды сопроцессора с их кратким описанием.

**Таблица 5.1. Набор команд сопроцессора**

Команда	Описание
<b>Команды передачи данных</b>	
FBLD	Загрузка BCD-числа (Binary coded decimal, двоично-десятичное число)
FBSTP	Сохраняет BCD-число и удаляет его из стека
FILD	Загружает целое число
FIST	Сохраняет целое число
FISTP	Сохраняет целое число и удаляет его из стека
FLD	Загружает действительное число
FSTP	Сохраняет действительное число и удаляет его из стека
FXCH	Обменивает местами два элемента стека
<b>Арифметические команды</b>	
FABS	Вычисляет абсолютное значение
FADD	Суммирует действительные числа

Команда	Описание
FIADD	Суммирует <b>целые</b> числа
FADDP	Суммирует действительные числа и удаляет результат из стека
FCHS	Изменяет знак числа
FDIV	Делит действительные числа
FIDIV	Делит <b>целые</b> числа
FDIVP	Делит <b>действительные</b> числа и удаляет результат из стека
FDIVR	Делит действительные числа с обратным порядком делимого и делителя
FIDIVR	Делит <b>целые</b> числа с обратным порядком делимого и делителя
FDIVRP	Делит действительные числа с обратным порядком делимого и делителя и удаляет результат из стека
FMUL	Умножает действительные числа
FIMUL	Умножает <b>целые</b> числа
FMULP	Умножает действительные числа и удаляет результат из стека
FPREM	Вычисляет частичный остаток
FPREM1	Вычисляет частичный остаток с использованием формата IEEE
FRNDINT	<b>Округляет</b> операнд до целого числа
FSCALE	Умножает на степень 2
FSUB	Вычитает действительные числа
FISUB	Вычитает <b>целые</b> числа
FSUBP	Вычитает действительные числа и удаляет результат из стека
FSUBR	Вычитает действительные числа с обратным порядком вычитаемого и уменьшаемого
FISUBR	Вычитает <b>целые</b> числа с обратным порядком вычитаемого и уменьшаемого
FSUBRP	Вычитает действительные числа с обратным порядком вычитаемого и уменьшаемого и удаляет результат из стека
FSQRT	Вычисляет <b>квадратный</b> корень
EXTRACT	Выделяет показатель степени и значение действительного числа
<b>Трансцендентные функции (все углы — в радианах)</b>	
F2XM1	Вычисляет значение $(2^x - 1)$
FCOS	Вычисляет косинус
FPATAN	Вычисляет арктангенс
FPTAN	Вычисляет тангенс
FSIN	Вычисляет синус
FSINCOS	Вычисляет синус и косинус
FYL2X	Вычисляет выражение $y \cdot \log_2 x$
FYL2XP1	Вычисляет выражение $y \cdot \log_2(x+1)$

Команда	Описание
<b>Константы</b>	
FLD1	Загружает 1.0
FLDL2E	Загружает $\log_2 e$
FLDL2T	Загружает $\log_2 10$
FLDLG2	Загружает $\log_{10} 2$
FLDPI	Загружает $\pi$
FLDZ	Загружает 0
<b>Операторы сравнения</b>	
FCOM	Сравнивает действительные числа
FCOMP	Сравнивает действительные числа и удаляет данные из стека
FCOMPP	Сравнивает действительные числа и дважды удаляет данные из стека
FICOM	Сравнивает целые числа
FICOMP	Сравнивает целые числа и удаляет данные из стека
FTST	Сравнивает вершину стека с нулем
FUCOM	Выполняет неупорядоченное сравнение
FUCOMP	Выполняет неупорядоченное сравнение со снятием со стека
FUCOMPP	Выполняет неупорядоченное сравнение с двумя снятиями со стека
FXAM	Проверяет значение $ST(0)$ и помещает результат в регистр условия
<b>Управляющие команды</b>	
FCLEX	Очищает все немаскированные исключения с плавающей точкой
FNCLEX	Очищает все исключения
FDECSTP	Уменьшает указатель стека
FFREE	Очищает элемент стека, как если бы он был удален из стека
FINCSTP	Увеличивает указатель стека
FINIT	Инициализирует FPU и проверяет наличие исключений
FNINIT	Инициализирует FPU без проверки наличия исключений
FLDCW	Загружает управляющее слово
FLDENV	Загружает окружение FPU
FNOP	Эквивалент NOP
FRSTOR	Восстанавливает состояние FPU из данной области памяти
FSAVE	Сохраняет состояние FPU в области памяти, проверяя исключения
FNSAVE	Сохраняет состояние FPU в области памяти без проверки исключений
FSTCW	Сохраняет управляющее слово с проверкой исключений
FNSTCW	Сохраняет управляющее слово без проверки исключений

Команда	Описание
FSTENV	Сохраняет окружение с проверкой исключений
FNSTENV	Сохраняет окружение без проверки исключений
FSTSW	Сохраняет слово состояния с проверкой исключений
FNSTSW	Сохраняет слово состояния без проверки исключений
FSTSW AX	Сохраняет слово состояния в AX с проверкой исключений
FNSTSW AX	Сохраняет слово состояния в AX без проверки исключений
WAIT	Приостанавливает процессор до завершения операции сопроцессора

Поскольку команд очень много, привести пример для каждой из них — нереальная задача. Поэтому в табл. 5.2 приведен общий формат большинства команд. Используя эту таблицу, **вы** сможете определить, какие операнды корректны для той или иной команды.

**Таблица 5.2. Форматы операндов сопроцессора**

Формат команды	Синтаксис операндов	Влечет использование
Классический	<i>Finstruction</i>	ST, ST(1)
С памятью	<i>Finstruction</i> memory	ST
С регистром	<i>Finstruction</i> ST(n), ST <i>Finstruction</i> ST, ST(n)	Нет
С регистром со снятием	<i>Finstruction</i> PST(n), ST	Нет

**НА ЗАМЕТКУ**

Все команды сопроцессора начинаются с буквы F. В случае работы с регистром со снятием со стека команды заканчиваются буквой P.

## Классический формат команд

Классический формат рассматривает стек FPU как обычный классический стек; все операции **обращаются** к вершине стека ST(0) и второму элементу стека ST(1). Например, суммирование выполняется при **помощи** команды FADD, результатом которой является сложение  $ST(0) = ST(0) + ST(1)$ . Таким образом, **значение** на вершине стека становится равным сумме значений двух верхних элементов стека, как показано на рис. 5.27. В целом команда **использует** элементы стека ST(0) и ST(1), если она работает с двумя **операндами**, и элемент ST(0), если с **одним**. При использовании двух операндов обычно верхний операнд снимается со стека, а результат операции помещается на вершину стека.

## Формат работы с памятью

Формат команд, работающих с памятью, подобен классическому в том, что стек сопроцессора рассматривается как классический стек. Однако данный формат позволяет вносить в стек данные из памяти и переносить их из стека в **память**. Например, можно загрузить в вершину стека операнд из памяти при помощи команды

FLD *память*

Снять операнд с вершины стека и перенести его в память можно при **помощи** команды

FSTP *память*

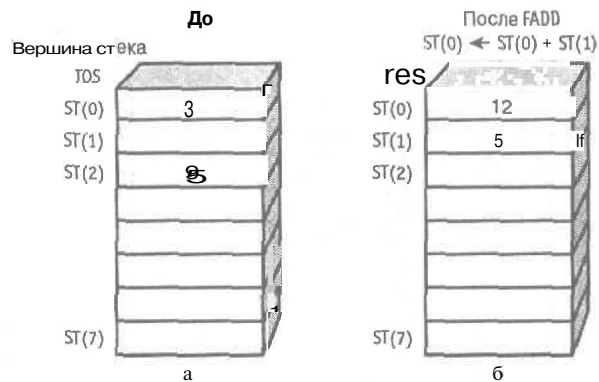


Рис. 5.27. Пример классической стековой команды

Результат работы приведенных команд показан на рис. 5.28.

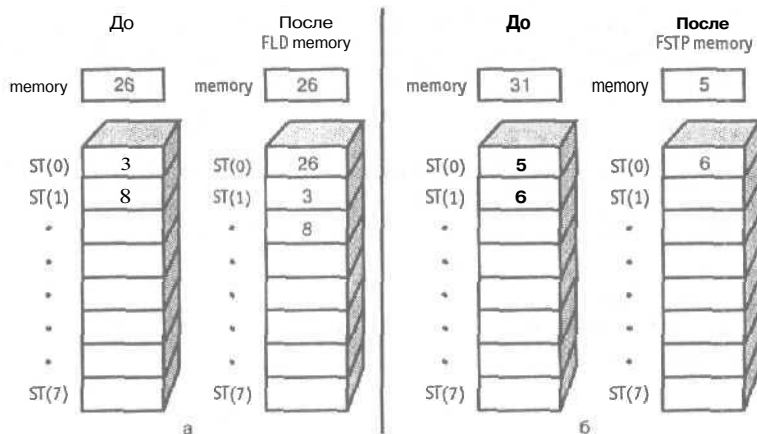


Рис. 5.28. Пример команд сопроцессора, работающих с памятью

## Формат работы с регистрами

Данный формат команд позволяет явно указывать регистры (элементы стека), к которым происходит обращение. Например, мы можем сложить значения из регистров ST(0) и ST(4). Команда

**FADD ST(0), ST(4)**

суммирует значения из указанных регистров и помещает результат на вершину стека, как показано на рис. 5.29. Если команда имеет два операнда, один из них должен быть вершиной стека ST(0) (или просто ST).

## Формат работы с регистрами и снятие со стека

Этот формат команды идентичен только что рассмотренному формату работы с регистрами, но после того, как результат вычислений помещается в стек, выполняется однократное снятие данных со стека.

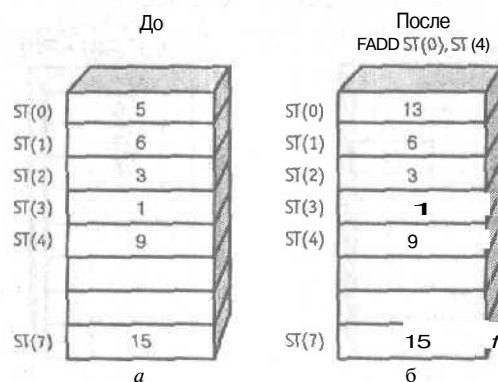


Рис. 5.29. Пример команды, работающей с регистрами

## Примеры команд сопроцессора

Неплохо было бы рассмотреть все команды сопроцессора, но у нас просто нет такой возможности. Поэтому мы рассмотрим только некоторые, наиболее характерные команды из полного набора. Это команды FLD, FST, FADD, FSUB, FMUL и FDIV.

### FLD

Все формы данной команды предназначены для загрузки действительного (FLD) или целого (FILD) числа или константы типа  $k$  (FLDPI) в стек. Команда имеет следующие форматы.

*Загрузка действительного числа*

FLD *op1*  
 32-битовое действительное число  
 64-битовое действительное число  
 80-битовое действительное число  
 ST(*n*)

*Загрузка целого числа*

FILD *op1*  
 16-битовое целое число  
 32-битовое целое число  
 64-битовое целое число

*Загрузка константы*

FLD*con*

где *con* — суффикс, определяющий загружаемую константу (см. табл. 5.1).

Рассмотрим пару конкретных примеров. Для загрузки действительного 32-битового числа из памяти мы используем команду

FLD *memory32*

после выполнения которой вершина стека сопроцессора ST(0) будет содержать значение, хранившееся в памяти (рис. 5.30а).

Для загрузки стандартного 16-битового целого числа в стек сопроцессора применяется команда

FILD *memory16*

после выполнения которой на вершине стека окажется 16-битовое целое число в формате IEEE (рис. 5.30б).

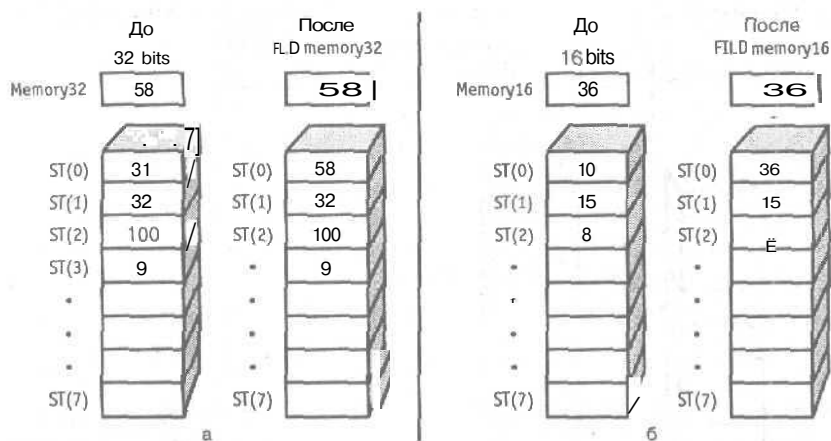


Рис. 5.30. Примеры работы команды загрузки

## FST

Команда FST используется для сохранения действительных (FST) и целых (FIST) чисел, находящихся на вершине стека ST(0). Суффикс P, прибавленный к команде, означает, что после сохранения значение снимается со стека. Команда имеет *следующие* форматы.

*Сохранение действительного числа*

FST op1  
 32-битовое действительное число  
 64-битовое действительное число  
 ST(n)

*Сохранение и снятие со стека действительного числа*

FSTP op1  
 32-битовое действительное число  
 64-битовое действительное число  
 ST(n)

*Округление и сохранение числа с вершины стека*

FIST op1  
 16-битовое целое число  
 32-битовое целое число  
 64-битовое целое число

*Округление, сохранение и снятие числа с вершины стека*

FISTP op1  
 16-битовое целое число  
 32-битовое целое число  
 64-битовое целое число

В качестве источника всегда используется вершина стека, так что передаваемый операнд op1 является указанием, где следует сохранить число с вершины стека.

Для того чтобы сохранить вершину стека в обычном числе с плавающей точкой в памяти, используется команда **FST memory32** (рис. 5.31а). При выполнении команды **FSTP memory32** элемент с вершины стека переносится в область памяти (рис. 5.31б). Если же вы хотите сохранить элемент с вершины стека в целом 32-битовом числе, то воспользуйтесь командой **FIST memory32** (рис. 5.31в).

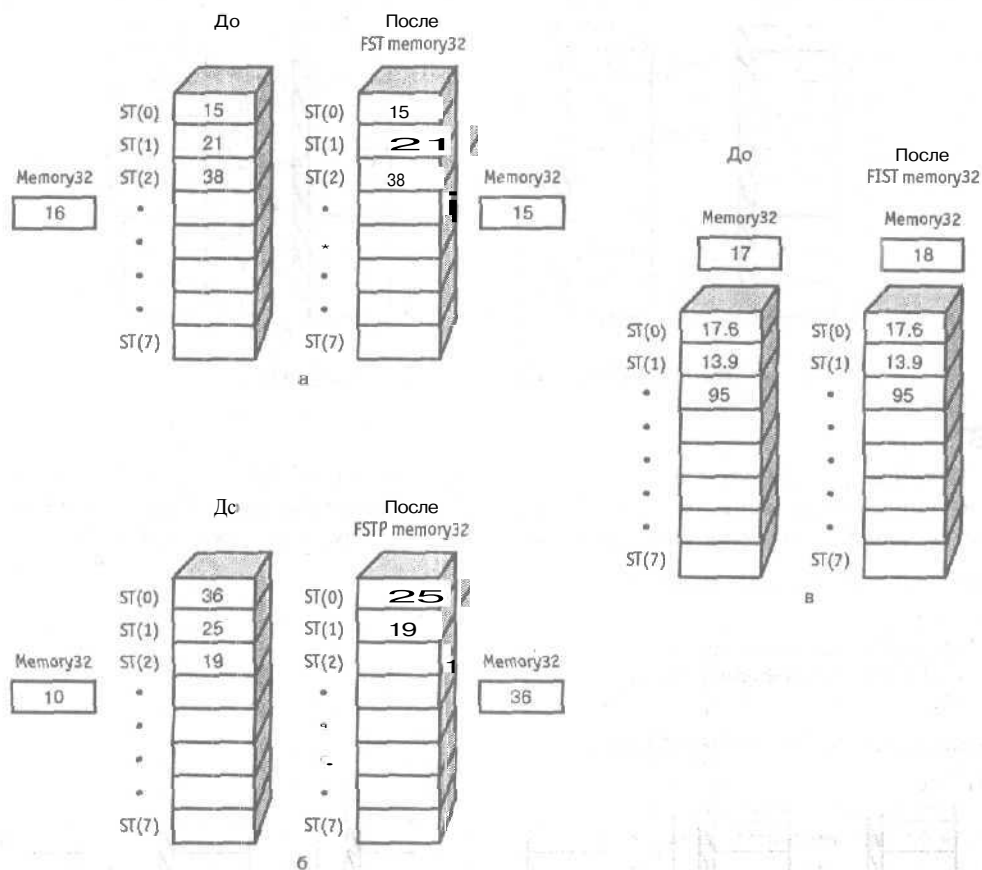


Рис. 5.31. Примеры команды сохранения

## FADD

Данная команда суммирует два действительных (FADD) или целых (FIADD) числа. К команде может быть добавлен суффикс **P**, который снимает элемент с вершины стека после выполнения сложения (эта возможность не поддерживается в версии для целых чисел). Команда **FADD** имеет следующие форматы.

Суммирование **ST(0)** и **ST(1)** со сдвигом стека

FADD

Суммирование с числом на вершине стека и помещением результата в **ST(0)**

FADD op1  
memory32

memory64  
ST(n)

Один из операндов — вершина стека, но результат может размещаться в другом элементе стека

FADD op1, op2  
ST(n), ST  
ST, ST(n)

После суммирования элемент ST(0) снимается со стека

FADDP opl, op2  
ST(n), ST  
ST, ST(n)

Целое число преобразуется в действительное и прибавляется к числу на вершине стека

FIADD opl  
memory16  
memory32

Теперь рассмотрим несколько конкретных примеров. Начнем со сложения двух верхних элементов стека. Для этого можно использовать команду

FADD ST, ST(1)

Результат ее выполнения показан на рис. 5.32а. Как видите, вычисленная сумма помещается на вершину стека.

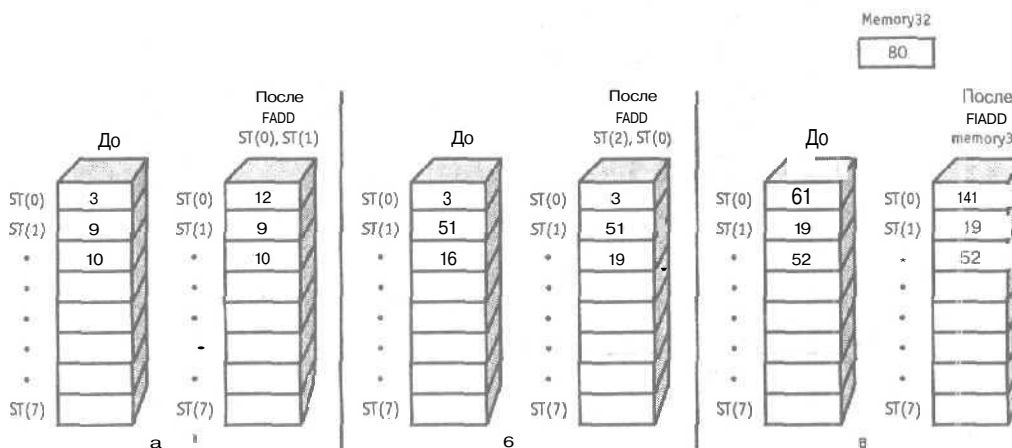


Рис. 5.32. Примеры суммирования чисел сопроцессором

Однако если вы воспользуетесь командой

FADD ST(2), ST

то сумма ST(2) и ST будет размещена в элементе ST(2), который в данном случае является целевым. Элемент ST(0) остается неизменным (рис. 5.32б).

Рассмотрим команду сложения чисел. Пусть, например, нам надо прибавить к числу на вершине стека 32-битовое целое число, находящееся в памяти. Для этого можно использовать команду

FIADD memory32

Значение, находящееся в памяти (memory32), преобразуется в действительное число и прибавляется к числу, находящемуся на вершине стека. Результат помещается на вершину стека (рис. 5.32в). Однако если мы захотим получить значение с вершины стека, то получим действительное, а не целое число. Для получения целого числа надо воспользоваться командой FISTP. Давайте соберем все рассмотренные команды в одном фрагменте кода и рассмотрим его работу шаг за шагом на диаграмме на рис. 5.33.

```
float fvalue_1 = 10.2;
float fvalue_2 = 50.1;
int ivalue_1 = 0;
```

```
_asm {
    FLD fvalue_1    // Загрузка первого 4-байтового числа
    FADD fvalue_2    // Загрузка и прибавление второго
                    // 4-байтового числа
    FISTP ivalue_1   // Сохранение результата в 4-байтовом
                    // целом числе
} // asm
```

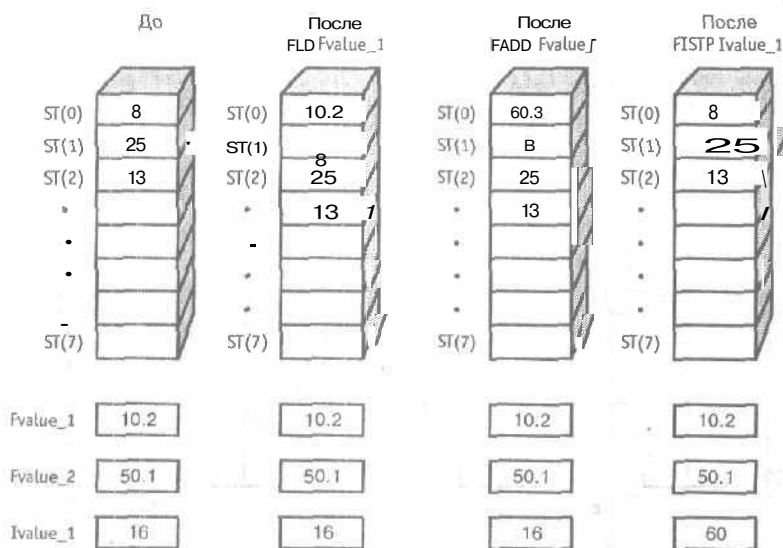


Рис. 5.33. Выполнение демонстрационного фрагмента кода

## FSUB

**Вычитание ST(1)-ST(0) со сдвигом стека**

FSUB

Вычитание операнда из числа на вершине стека и помещение результата в ST(0)

```
FSUB op1
    memory32
    memory64
    ST(n)
```

Один из операндов — вершина стека, но результат (op1-op2) может размещаться в другом элементе стека

**FSUB** op1, op2  
ST(n), ST  
ST, ST(n)

После вычитания элемент ST(0) снимается со стека

**FSUBP** opl, op2  
ST(n), ST  
ST, ST(n)

Целое число преобразуется в действительное и вычитается из числа на вершине стека

**FISUB** opl  
memory16  
memory32

Вычитание работает так же, как и сложение, за исключением того, что здесь более важен порядок операндов, поскольку  $a - b \neq b - a$ . Например, при выполнении команды **FSUB memory32** содержимое ячейки памяти вычитается из значения на вершине стека, т.е.  $ST(0) = ST(0) - \text{memory32}$ , как показано на рис. 5.34а.

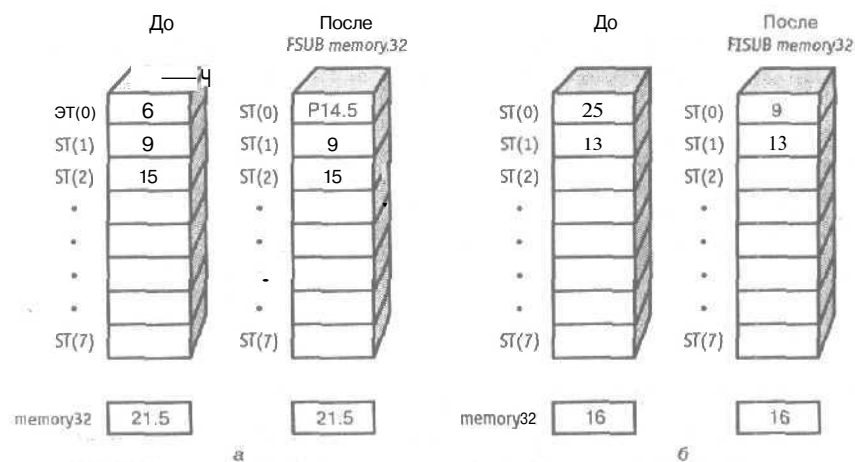


Рис. 5.34. Примеры команд вычитания

В качестве другого примера вычтем целое число из действительного числа на вершине стека (скажем, для вычисления десятичной части). Для этого можно воспользоваться командой **FISUB memory16**, после выполнения которой на вершине стека окажется значение разности ST(0) и преобразованного в действительное целого числа из memory16, как показано на рис. 5.34б.

## FMUL

Команда **FMUL** очень похожа на команду **FADD**. Она применяется для умножения действительных чисел (**FMUL**) и целого и действительного чисел (**FIMUL**), и, как обычно, суффикс **P** означает снятие значения со стека после выполнения операции.

Умножение  $ST(Q) * ST(1)$  со сдвигом стека

**FMUL**

Умножение числа на вершине стека на операнд и помещение *результата* в ST(0)

```
FMUL op1
    memory32
    memory64
    ST(n)
```

Один из операндов — вершина стека, но результат может размещаться в другом элементе стека

```
FMUL op1, op2
    ST(n), ST
    ST, ST(n)
```

После умножения элемент ST(0) снимается со стека

```
FMULP op1, op2
    ST(n), ST
    ST, ST(n)
```

Целое число преобразуется в действительное и умножается на число на вершине стека

```
FIMUL op1
    memory16
    memory32
```

Мне уже не хватает идей для примеров... Давайте вычислим 5! — значение, равное  $5*4*3*2*1$ . Ниже представлен фрагмент кода, который делает это.

```
_asm {
    FLD f_1 // Загружаем 1.0 в стек
    FLD f_2 // Загружаем 2.0 в стек
    FLD f_3 // Загружаем 3.0 в стек
    FLD f_4 // Загружаем 4.0 в стек
    FLD f_5 // Загружаем 5.0 в стек
    FMUL // 5*4
    FMUL // 5*4*3
    FMUL // 5*4*3*2
    FMUL // 5*4*3*2*1
    // Вершина стека ST(0) содержит значение 5*4*3*2*1
} // asm
```

На рис. 5.35 показано пошаговое выполнение приведенного фрагмента кода.



Рис. 5.35. Пошаговое вычисление факториала

## FDIV

Последний класс команд, которые мы рассмотрим, — это команды деления. Они очень похожи на команды вычитания, и в них точно так же важен порядок указания операндов.

*Деление ST(1)/ST(0) со сдвигом стека*

### FDIV

*Деление числа на вершине стека на передаваемый операнд и помещение результата в ST(0)*

```
FDIV op1
      memory32
      memory64
      ST(n)
      ST(n)
```

*Один из операндов — вершина стека, но результат (op1/op2) может размещаться в другом элементе стека*

```
FDIV op1, op2
      ST(n), ST
      ST, ST(n)
```

*После деления элемент ST(0) снимается со стека*

```
FDIVP op1, op2
      ST(n), ST
      ST, ST(n)
```

*Целое число преобразуется в действительное и на него делится число на вершине стека*

```
FIDIV op1
      memory16
      memory32
```

В качестве примера проведем вычисления по формуле  $\text{скорость} = \text{расстояние} / \text{время}$ . Вычислим, сколько времени потребуется ракете, чтобы долететь до Луны.

```
// Расстояние до Луны: 384000 км
float f_dist = 384000.0;
```

```
// Скорость корабля 38400 км/ч
float f_vel = 38400;
```

```
// Время полета
float f_time;
```

```
_asm {
    FLD f_dist // Загружаем расстояние
    FDIV f_vel // Делим на скорость
    FSTP f_time // Сохраняем результат в переменной
                // f_time и снимаем его со стека
} // asm
```

```
printf("\ntime = %f", f_time);
```

#### СОВЕТ

Обратите внимание, как легко осуществляется вставка ассемблерных команд в код на C/C++.

## Замечания по использованию математической библиотеки

Использование математической библиотеки по сути ничем не отличается от использования других библиотечных модулей. Однако есть некоторые моменты, которые не следует забывать. Во-первых, наша математическая библиотека опирается на часть кода из файлов `T3DLIB1.CPP|H`, поэтому вы должны убедиться, что в вашу программу, использующую математическую библиотеку, включен файл `T3DLIB1.H`, а в проект подключен файл `T3DLIB1.CPP`. Кроме того, библиотека `T3DLIB1.CPP|H` основана на DirectX 8.0+, так что вы должны включить заголовочный файл `DDRAW.H`. Итак, при построении обычного консольного приложения вам потребуются следующие файлы.

Файлы с исходными текстами

- `T3DLIB1.CPP`
- `T3DLIB4.CPP`

### Заголовочные файлы

- `T3DLIB1.H`
- `T3DLIB4.H`
- `DDRAW.H`

### Библиотечные файлы

- Отсутствуют

Конечно, вы должны включить также все обычные заголовочные файлы, которые нужны для создания приложения. Не забывайте о конструкции `tfdefirte INITGUID`, без которой компоновщик не сможет найти `DirectDraw`, и о взаимозависимости библиотек. При разработке математической библиотеки мои тестовые программы начинались со следующего кода, в котором учтено все сказанное выше.

```
#define WIN32_LEAN_AND_MEAN
```

```
#ifndef INITGUID
```

```
#define INITGUID // Необходимо для библиотеки DXGUID.LIB
```

```
#endif
```

```
#include <windows.h> // Включение функциональности Windows
```

```
#include <windowsx.h>
```

```
#include <mmsystem.h>
```

```
#include <objbase.h>
```

```
#include <iostream.h> // Стандартные возможности C/C++
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
#include <malloc.h>
```

```
tfinclude <memory.h>
```

```
#include <string.h>
```

```
flinclude <stdarg.h>
```

```
ffinclude <stdio.h>
```

```
tfinclude <math.h>
```

```
#include <io.h>
```

```
#include <fcntl.h>
#include <direct.h>
#include <wchar.h>

#include <ddraw.h> // Требуется библиотекой T3DLIB1.H
#include "T3DLIB1.H" // Используется библиотекой T3DLIB4
#include "T3DLIB4.H"
```

Ясно, что приложения, которые используют игровую библиотеку целиком, должны включать следующие файлы.

Исходные файлы

- T3DLIB1.CPP — поддержка графики и общие утилиты
- T3DLIB2.CPP — поддержка ввода-вывода
- T3DLIB3.CPP — поддержка звука и музыки
- T3DLIB4.CPP — математическая библиотека

Заголовочные файлы

- T3DLIB1.H
- T3DLIB2.H
- T3DLIB3.H
- T3DLIB4.H

Библиотечные файлы

- DDRAW.LIB — DirectDraw
- DSOUND.LIB — DirectSound
- DINPUT.H — DirectInput
- DINPUT8.H — DirectInput
- DMKSCtrl.H — DirectMusic
- DMUSICI.H — DirectMusic
- DMUSICC.H — DirectMusic
- DMUSICF.H — DirectMusic
- WINMM.LIB — Расширения мультимедиа Windows (только для пользователей Visual C++)

Поскольку у нас появился новый библиотечный файл, мы должны обновить шаблон консоли игры. Теперь это файл с именем T3D\_CONSOLE3.CPP, а отличается он от своего предшественника лишь наличием строки

```
#include "T3DLIB4.h"
```

## Замечания об оптимизации

Код в этой главе перенасыщен различными алгоритмами. Так, при вычислении расстояний использовался ряд Тейлора, выделение множителей — при вычислении определителей и т.д. Все это делалось с одной целью — ускорить работу функций. Такое ускорение можно делать на низком уровне, используя разворачивание циклов, алгебраические упрощения вычислений и т.п. Вы, наверное, обратили внимание, что я часто нахожу обратное значение чего-либо, а потом выполняю умножение на эту величину? Дело в том, что умножение выполняется существенно быстрее деления.

Но, как я уже говорил, главная оптимизация — это оптимизация на уровне алгоритмов и свойств исходных данных. Например, нет смысла выполнять умножение разреженных матриц (т.е. матриц, у которых большинство элементов — нулевые) по стандартным формулам, так как большую часть времени вы будете перемножать и складывать нули, даже если вы будете делать это с привлечением всех своих знаний по ассемблеру. И только исчерпав возможности алгоритмической оптимизации, можно приступать к оптимизации на более низком уровне. Но я буду удивлен, если вам удастся существенно улучшить код, генерируемый хорошим оптимизирующим компилятором C/C++.

## Резюме

Если вы внимательно и полностью прочли эту главу, то вам наверняка снятся **векторные** кошмары. Но зато вы получили неплохое подспорье для работы над играми в виде математической библиотеки. Пожалуй, в главе слишком много кода — но ведь это книга для программистов! Теперь, если вы захотите воспользоваться моими трудами — то перед тем, как начать писать собственный код для решения той или иной математической задачи, посмотрите, не решил ли уже я ее для вас.

Единственная проблема, возникновение которой в ближайших главах кажется мне неминуемой, — это вопрос о том, следует ли использовать трехмерные или четырехмерные однородные **координаты**. Не зря в библиотеке имеются функции для работы с матрицами самых разных размеров, поэтому проблема не будет такой уж значительной.

# ГЛАВА 6

## Введение в трехмерную графику

### В этой главе...

• Философия трехмерного игрового процессора	410
• Структура трехмерного игрового процессора	410
• Трехмерные системы координат	423
• Базовые трехмерные структуры данных	467
• Инструментарий трехмерного моделирования	483
• Загрузка внешних данных	484
• Основы преобразований твердых тел и анимации	502
• Обзор конвейера визуализации	507
• Типы трехмерных игровых процессоров	508
• Сборка игрового процессора	514

В этой главе будут рассмотрены различные темы, лежащие в основе трехмерной графики. Материал, представленный в данной главе, очень важен для целостного восприятия основных понятий трехмерной графики:

- философия трехмерного игрового процессора;
- структура трехмерного игрового процессора;
- трехмерные системы координат;
- основные структуры данных;
- преобразования и основы анимации;
- введение в конвейеры визуализации;

- типы трехмерных процессоров;
- инструментарий для трехмерной графики;
- загрузка данных.

## Философия трехмерного игрового процессора

Главная цель данной книги — обучить вас трехмерной графике, а не тому, как использовать конкретный API или ускорители. Я хочу, чтобы вы ясно представляли себе, что это означает. Эта книга дает вам все необходимые знания для создания своего собственного трехмерного игрового **процессора** — от представления данных до конечной растеризации. Так что если после прочтения этой книги вы получите работу в NVIDIA, ATI или подобной фирме — то сразу сможете создавать процессоры, API и работать с лежащими в их основе алгоритмами, которые затем будут реализованы аппаратно. С другой стороны, данная книга *не* учит вас использовать тот или иной трехмерный API, например, API DirectX или **OpenGL**. Это мощные средства, но если вы действительно хотите понимать трехмерную графику, то вы должны понимать, каким образом можно самостоятельно разработать DirectX или OpenGL. Моя задача — научить вас разработке программного трехмерного процессора на любой машине с линейной адресацией видеобuffers. Если вы будете в состоянии сделать это — разобраться в любом API для вас не представит никакого труда.

## Структура трехмерного игрового процессора

Поскольку нас интересует программирование игр, главная **цель** трехмерного игрового процессора — визуализация трехмерной игры. Это означает, что игровой процессор, физическая система, система искусственного интеллекта и прочие составные части игры должны быть тесно связаны с трехмерным **процессором**. В противном случае проблем у нас будет гораздо больше, чем их решений.

Таким образом, при разработке трехмерного процессора надо едва ли не в первую очередь думать о его взаимодействии с игровым процессором. На рис. 6.1 представлена примерная схема трехмерного игрового процессора, благодаря которой становится очевидно, что в трехмерную игру должны входить **следующие** составляющие.

- Трехмерный процессор
- Игровой процессор
- Система ввода и работы в сети
- Система анимации
- Система навигации и обнаружения столкновений
- Физический процессор
- Система искусственного интеллекта
- База данных трехмерных моделей и изображений

Как видите, есть много вещей, о которых надо не забыть! В простой двумерной игре обнаружение столкновений, система искусственного интеллекта и база данных игры были достаточно просты и понятны, но над реализацией всего перечисленного в трехмерном случае надо крепко поломать голову. Вначале подробнее ознакомимся с **различными** частями игрового процессора.

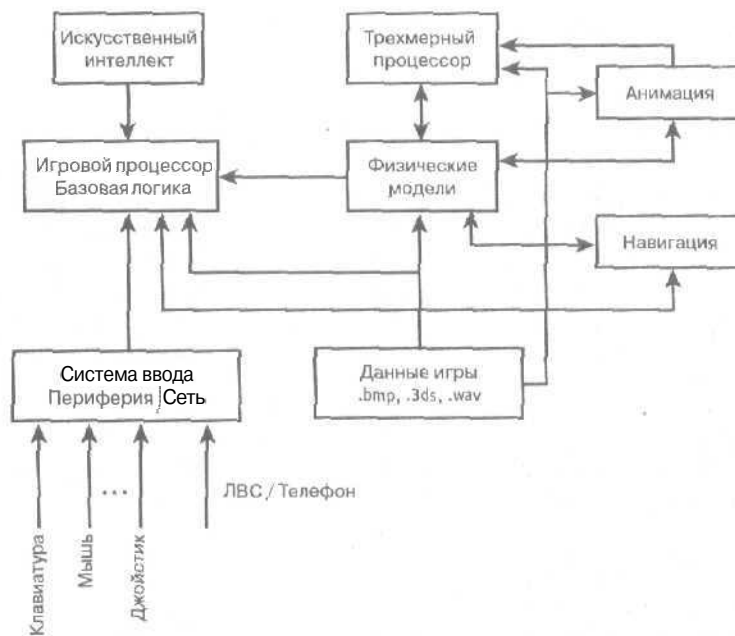


Рис. 6.1. Обобщенная структура трехмерного игрового процессора

## Трехмерный процессор

Трехмерный процессор представляет собой программное обеспечение, которое обрабатывает структуры данных трехмерного мира (включая источники освещения, действующие лица, общую информацию о состоянии игры и т.п.) и визуализирует игровой мир с точки зрения игрока или камеры. Решение этой задачи *относительно* простое и без изысков, если игровой процессор разрабатывается модульным, так что имеется не так много связей между трехмерным процессором и другими *игровыми* подсистемами. Одна из наибольших ошибок, которую можно сделать при написании трехмерного процессора, — это сделать его ответственным не только за трехмерную визуализацию. Например, попытка внести физическое моделирование в код *трехмерного* процессора наряду с кодом трехмерной графики является очень большой ошибкой! Другим примером может быть размещение в трехмерном *процессоре* сетевого перехвата для упрощения работы в сети.

### СОВЕТ

Трехмерный процессор решает *одну-единственную* задачу — трехмерную визуализацию.

## Игровой процессор

Игровой процессор трехмерной игры в большинстве случаев представляет собой *управляющий* модуль, который шлет команды всем другим подсистемам. Следовательно, игровой процессор следует разрабатывать так, чтобы он мог подключаться к любой *другой* подсистеме *игры*. Зачастую допускается *ошибка*, состоящая в том, что трехмерную игру начинают с разработки трехмерного процессора, а затем в некоторый момент процесса разработки превращают часть трехмерного процессора к подсистему управления. Это не лучший способ разработки *игры*. С самого начала надо разрабатывать модуль управления всеми остальными *модулями*, т.е. игровой *процессор*.

сор. Его можно представить также в виде контейнера, содержащего все остальные части игры.

## Система ввода и работы в сети

Система ввода-вывода и работы с сетью в трехмерной игре пользуется дурной славой самой проблемной подсистемы, и пользуется заслуженно. Проблема в том, что потенциально по сети должно пересылаться такое огромное количество данных, что это делает **работающую** сетевую трехмерную игру поистине чудом. На рис. 6.2 показаны две разные модели сетевых игр. Главное правило при создании сетевой трехмерной игры заключается в том, что в первую очередь надо думать об **игре**, а не о поддержке сети, и игра должна быть отделена от сети. Поддержка сети не может быть частью трехмерного процессора, искусственного интеллекта, физического моделирования — словом, это совершенно отдельный вопрос о том, какие структуры данных следует использовать, как обеспечить **синхронизацию**, причем все это должно быть согласовано с принципами самой игры. Поэтому вопросы сетевой **поддержки** и вопросы дизайна игры должны рассматриваться одновременно и с самого начала ее разработки.

### СОВЕТ

При написании двумерной игры вы можете попытаться добавить сетевую поддержку на заключительном этапе — правда, это приведет к резкому увеличению трудозатрат и появлению огромного количества уродливого кода. Но в трехмерной игре добавить сетевую поддержку на последнем этапе работы просто **невозможно**. Это решение должно приниматься изначально, причем от вас потребуются **серьезная** работа по созданию сетевой модели и ее всестороннему тестированию.

## Анимация

В трехмерных играх встречается анимация нескольких видов, а именно:

- простое движение;
- сложная анимация;
- анимация физических моделей.

### Простое движение

Простое движение обычно состоит из **перемещения** и поворотов объектов **игры**, как показано на рис. 6.3. Движение может **управляться** файлами данных, искусственным интеллектом, логической системой и т.п. В любом случае некая **управляющая** система определяет, каким образом будет перемещаться некоторые объекты.

**а) Модель равноправных компьютеров**



**За:** Простота реализации

**Против:** Трудность масштабирования

**Игроков:** 2-4

**б) Модель "клиент-сервер"**



**За:** Хорошая масштабируемость

**Против;** Трудность реализации

**Игроков:** 2-32 и больше

Рис. 6.2. Модели сетевых игр

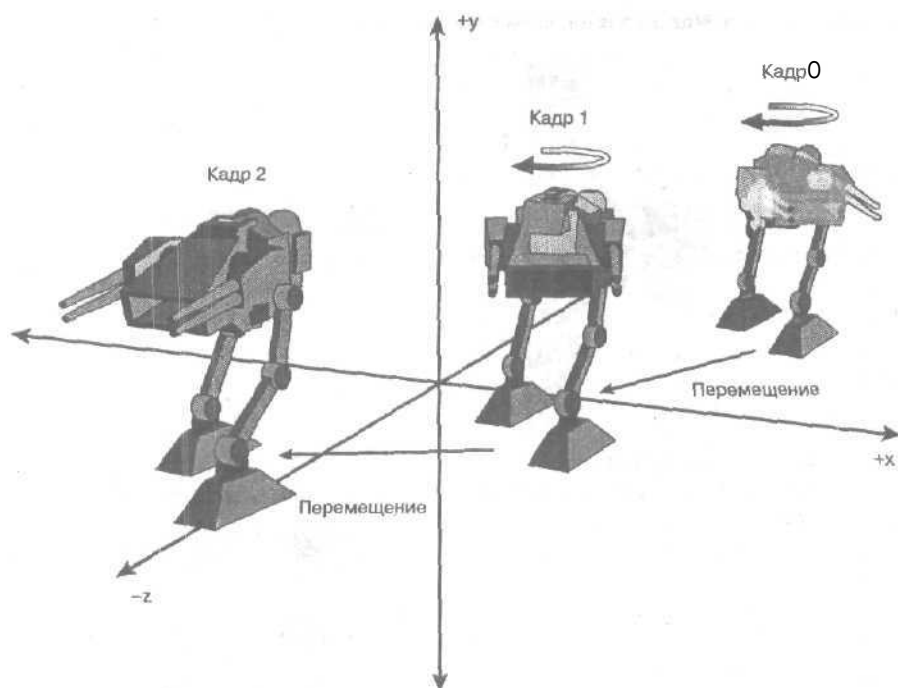


Рис. 6.3. Простое перемещение и поворот

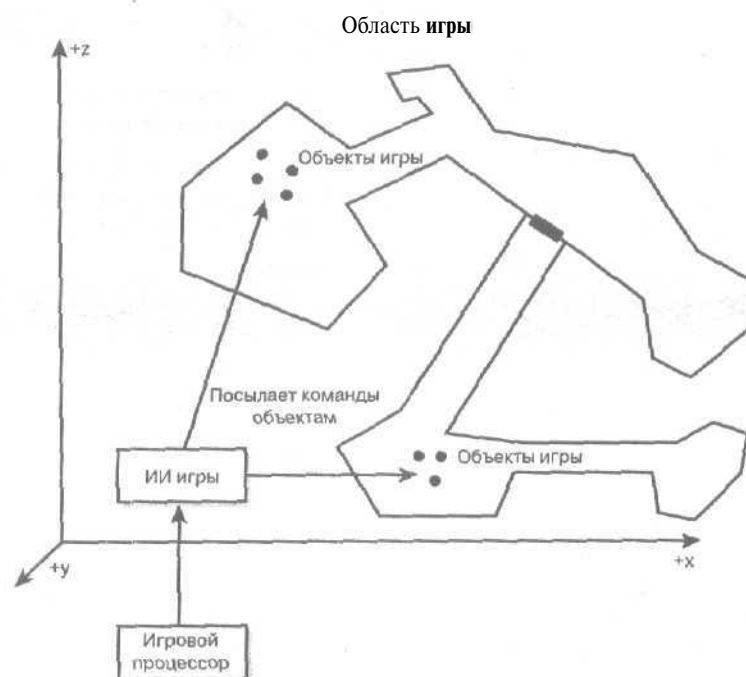


Рис. 6.4. Перемещение объектов системой искусственного интеллекта

Эти объекты двигаются в пределах игровой карты (рис. 6.4). Если система искусственного интеллекта имеет представление этой карты, она в состоянии обнаружить столкновения и не допустить прохода объектов сквозь стены. Вероятно, в реальной *игре* дело будет обстоять не так просто, и система искусственного интеллекта будет передавать команды не объектам непосредственно, а системе физического моделирования, которая, в свою очередь, будет определять перемещения объектов и возможность таковых перемещений. Хотя суть дела от этого, впрочем, не изменяется.

### Сложная анимация

Сложная анимация в трехмерных играх обычно означает, что у нас имеются *иерархически сочлененные* объекты, причем со звеньями, которые должны двигаться относительно друг друга. Пример такого объекта представлен на рис. 6.5, где изображен некий *робототанк*. Для простоты будем считать, что танк состоит из двух частей — тела и пушки. Пушка может вращаться относительно вертикальной оси, проходящей через тело, но при этом она, естественно, должна оставаться прикрепленной к танку. Возникает вопрос: как должна управляться пушка — искусственным интеллектом или на *основании* некоторых данных? Другими словами, предположим, что мы хотим, чтобы танковая пушка *выполнила* некоторую задачу. Может оказаться достаточно сложно сделать это при помощи искусственного интеллекта, но зато очень легко — при помощи команд анимации. Давайте остановимся на этом подробнее.

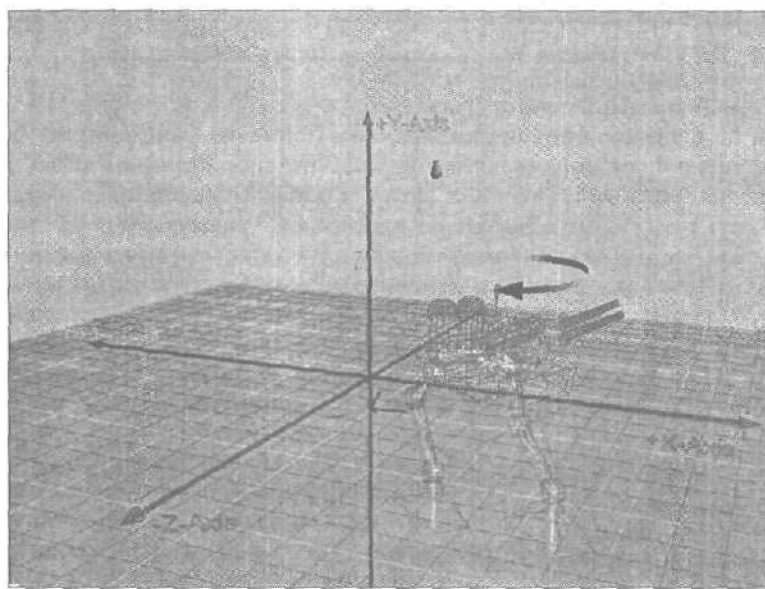


Рис. 6.5. Анимированная составная трехмерная модель

Существует два способа анимации трехмерного составного объекта. Первый способ состоит в анимации с использованием инструментария трехмерного моделирования, импортировании сеток и анимации как функции времени — просто загрузки сетки 1, затем сетки 2, сетки 3 и т.д. Эта технология похожа на стандартную растровую двумерную анимацию, но с тем отличием, что теперь трехмерная модель представлена трехмерными данными. Этот метод предусматривает потребление большого количества памяти, но зато очень прост в реализации. Пример его использования показан на рис. 6.6. Здесь мы

моделируем танк с пушкой, повернутой под тремя разными углами, и сохраняем полученные сетки. Затем для анимации танка мы загружаем сетки и визуализируем их по одной. Такая технология использована в играх *Quake* и *Quake II*.

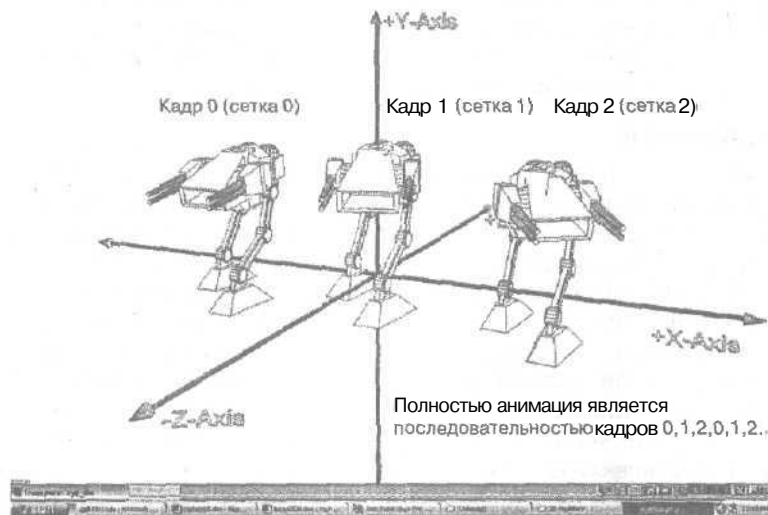


Рис. 6.6. Использование нескольких кадров для создания анимации

Другой способ анимации трехмерных объектов состоит в использовании *данных движения*. Один из наиболее популярных форматов *движения* — формат BioVision (.BVH), маленький фрагмент которого приведен ниже. Детальное описание этого формата вы можете найти на прилагаемом компакт-диске, в файле **BIOVISION.DOC** в каталоге данной главы. Данные в этом формате указывают процессору, каким образом движутся связанные между собой части *объекта*, и процессор в соответствии с этими данными выполняет изменение внешнего вида объекта, перемещая его составные части. Такая методика гораздо гибче, поскольку после того, как создан соответствующий процессор, вы уже можете моделировать любое движение — от походки до танца. Эта технология использована в играх *Quake Arena*, *Doom III*, *Halo* и других. Если вы хотите приобрести данные движений, обратитесь к BioVision по адресу <http://www.biovision.com>.

```
HIERARCHY
ROOT Hips
{
  OFFSET 20 0.00 0.00
  CHANNELS 6 Xposition Yposition Zposition Zrotation
  Xrotation Yrotation
  JOINT LeftHip
  {
    OFFSET 3.430000 0.000000 0.000000
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftKnee
    {
      OFFSET 0.000000 -18.469999 0.000000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftAnkle
      {
```



ние, то система анимации должна передать проверки системе физического моделирования. Однако если физики в вашей игре немного или нет совсем — вероятно, проверку возможности выполнения движений лучше оставить подсистеме анимации.

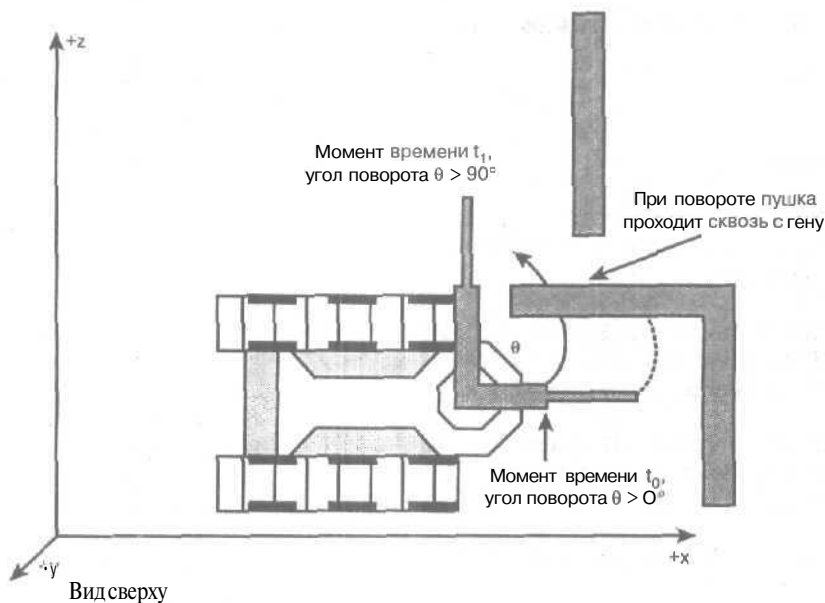


Рис. 6.7. Физически невозможная ситуация при выполнении нефизической анимации

## Анимация физических моделей

Это наиболее сложная анимация, которая управляется законами физики. Но такая анимация выглядит наиболее реалистично. В случае анимации физических моделей подсистемы анимации и физического моделирования тесно связаны между собой. Новые положения и ориентация объектов в пространстве вычисляются в ходе игры в реальном времени в соответствии с физическими законами. Например, брошенная граната не просто летит в определенном направлении, но может удариться о стены, отражаться, падать под действием силы **тяжести** — словом, вести себя как настоящий реальный объект. Но какой бы не была подсистема анимации, она не занимается **визуализацией** — для этого есть **соответствующая** подсистема, которая должна знать лишь список составляющих объекты многоугольников и не более того.

## Система навигации и обнаружения столкновений

Мы не раз уже говорили о системе физического моделирования, которая тесно связана с остальными частями трехмерной игры. Однако бывает и так, что настоящее физическое моделирование совершенно излишне и достаточно простейшего обнаружения столкновений и системы **навигации**. Конечно, эту функциональность можно назвать физическим моделированием, но она не основывается на каких-либо физических законах и величинах, так что с тем же успехом можно назвать автомобилем детскую игрушку. Для примера предположим, что мы создаем игру в стиле *Doom*, в которой физика нам совершенно ни к чему: все персонажи **анимируются** системой искусственного интеллекта с применением **предварительно** созданных сеток. Все, что нам надо в таком случае, — это обнаружение столкновений объектов со стенами и между собой (рис. 6.8).

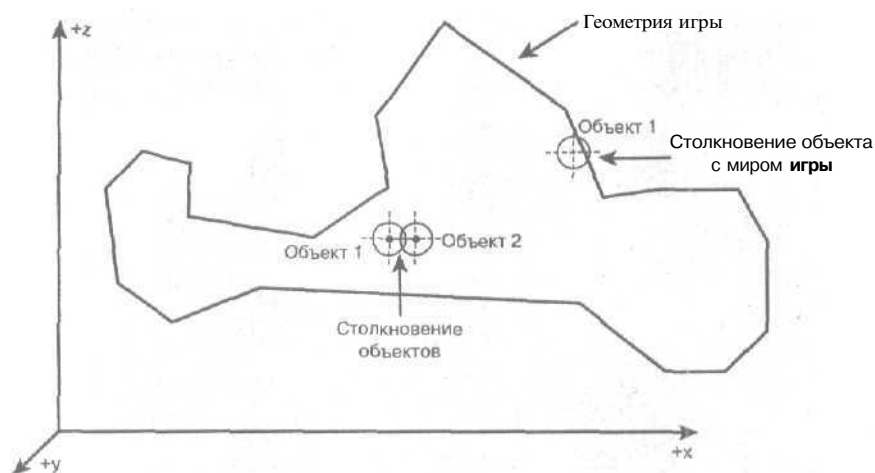


Рис. 6.8. Столкновения объектов со стенами и между собой

Таким образом, нам достаточно простейшей системы обнаружения столкновений, которая характеризуется обратной связью с подсистемой анимации (и/или подсистемой искусственного интеллекта). Однако при не очень сложной системе обнаружения столкновений можно получить определенное проникновение объектов друг в друга, перемещение персонажей ниже (или выше) уровня пола и тому подобные "проколы".

## Физический процессор

Физическая подсистема в игре может быть как очень простой, например, ограниченной обнаружением столкновений, так и очень сложной, основанной на реальном физическом моделировании. Физика в игровом процессоре никак не связана с визуализацией трехмерной графики. Все, чем занимается подсистема физического моделирования, — это управление объектами и их реакцией на действия (или отсутствие действий) со стороны других объектов и выполнение ими некоторых действий на основе реализованных физических моделей. Так, простейшая физическая модель может обладать следующими возможностями:

- учет сил трения и ускорения;
- обработка упругих соударений;
- передача импульса.

Примером такой простейшей модели могут служить трехмерные гонки, представляющие собой плоскую трассу и несколько автомобилей (рис. 6.9). Каждый автомобиль можно представить в виде материальной точки с определенной массой (или четырех точек — по одной для каждого колеса). При движении автомобиля по трассе под управлением игрока или подсистемы искусственного интеллекта используется физическая модель, в соответствии с которой автомобиль удерживается на дороге исключительно благодаря силе трения. То есть при повороте движущегося автомобиля вычисляется центробежная сила, сравнивается с максимальной силой трения, а затем на основе полученных величин изменяется направление и скорость автомобиля. Столкновение автомобилей рассматривается как упругое соударение сфер с соответствующим обменом импульсами (моменты импульса для простоты не рассматриваются).

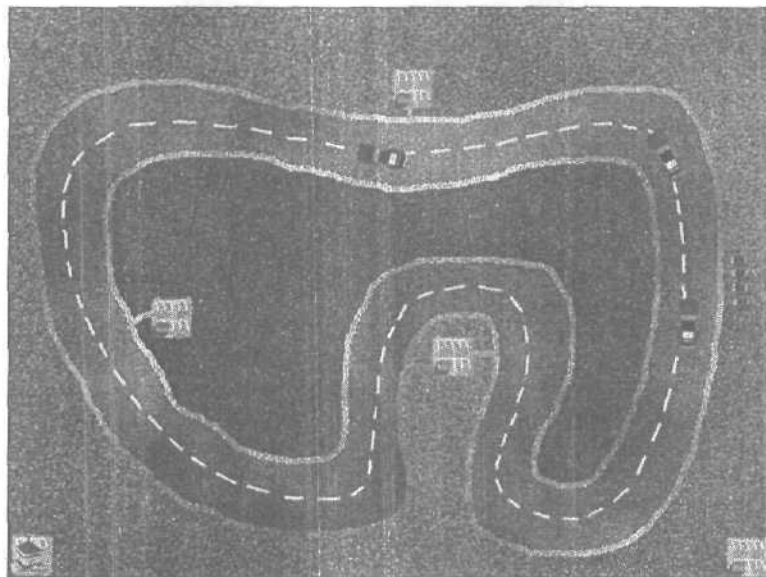


Рис. 6.9. Простейшие гонки на трассе

Такая физическая модель вполне адекватна для множества двумерных и трехмерных игр, но если вы хотите достичь большей реалистичности, то вы должны пойти дальше и учесть реальную геометрию объектов, физические законы их движения. Таким образом, движением объектов должна управлять подсистема физического моделирования, а не искусственный интеллект, оснащенный системой обнаружения столкновений и простейшим расчетом сил трения. В полной физической модели непрерывно контролируется состояние каждого объекта и учитываются все **воздействующие** на него силы. Интегрированием уравнений движения определяется положение каждого объекта как функции времени и приложенных к объекту сил.

Более подробно мы поговорим об этом в части книги, посвященной физическому моделированию. Единственное, что следует добавить, — при таком уровне физического моделирования трехмерная игра должна иметь подсистемы искусственного интеллекта и анимации, подключенные к физической подсистеме, в частности для того, чтобы искусственный интеллект мог управлять объектом, не требуя от него невозможного с точки зрения физики.

## Система искусственного интеллекта

Искусственный интеллект в трехмерной игре может быть на порядок сложнее, чем в двумерной, или того же уровня сложности — все зависит от самой игры. В трехмерности и состоит главная проблема искусственного интеллекта в трехмерной игре, когда **существенно** усложняются алгоритмы навигации, поиска **путей** и прочие условия, **зависящие** от размерности пространства. В трехмерных играх используются те же методы и алгоритмы, что и в двумерных — **конечные** автоматы, шаблоны, нечеткая логика, нейронные сети и т.п. — но их реализация усложняется из-за наличия третьего измерения.

Для примера **взгляните** на рис. 6.10. Здесь изображены двумерная и трехмерная среды из двух различных игр-«стрелялок» (*Valkyrie* и *Quake III*). Искусственный интеллект двумерного прокручивания легко реализуется благодаря простоте игровой среды. Объекты

представляют собой не более чем двумерные растровые анимации, управляемые шаблонами и конечными автоматами. Трехмерная игра работает примерно по тому же принципу, но ее написание на порядок сложнее. Дело в том, что все объекты здесь трехмерные, и даже обычное перемещение персонажей становится большой проблемой. Любое движение, анимация, логика действий персонажей должны быть приспособлены к внешней трехмерной среде.

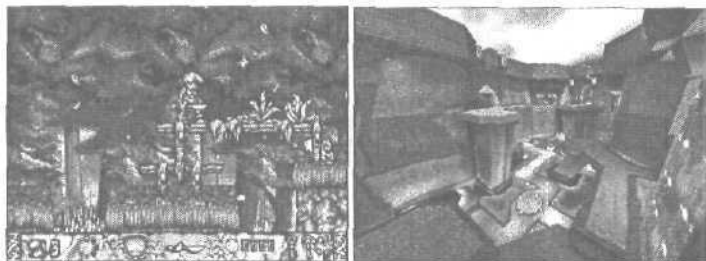


Рис. 6.10. Двумерная и трехмерная игровые среды

Все сказанное делает программирование искусственного интеллекта для трехмерной игры по-настоящему сложной задачей. Операции наподобие простого перемещения объекта из одной точки в другую в трехмерном мире могут оказаться непреодолимо сложными из-за геометрии мира игры. Может быть так, что для человека поиск простого пути оказывается делом одного взгляда, но для искусственного интеллекта такой поиск оказывается сложной трудоемкой задачей. Одним словом, искусственный интеллект в трехмерном мире должен быть "умнее" своего коллеги из мира двумерного. И наконец, подсистема искусственного интеллекта в трехмерной игре должна быть тесно связана с подсистемами анимации и физического моделирования, поскольку искусственный интеллект для своей работы в трехмерном мире должен получать массу информации об этом мире.

## База данных трехмерных моделей и изображений

Последней большой составляющей трехмерной игры являются данные. И этих данных в трехмерной игре действительно очень много! Это связано с самой природой трехмерности. Так, типичная трехмерная игра должна работать со множеством данных следующих типов:

- каркасы трехмерных объектов;
- двумерные текстуры и источники света;
- данные о трехмерном мире игры;
- данные о движении и анимации;
- карта игры.

Обычно наибольшее количество памяти требуется для хранения текстур и источников света, поскольку в игре могут использоваться сотни, если не тысячи текстур. Если работать с текстурами размером  $1024 \times 1024$  с 24-битовыми цветами, то каждая текстура потребует три мегабайта памяти. Понятно, что это — экзотика, и, как правило, в играх используются задаваемые пользователем размеры текстур и их качество, но, тем не менее, даже текстура размером  $256 \times 256$  в 8-битовом режиме требует 64 килобайта памяти. В нашей книге мы будем в основном иметь дело с текстурами размером от  $32 \times 32$  до  $128 \times 128$  — для простоты (и скорости — ведь мы используем программные решения), но, я думаю,

вы понимаете, что в реальных играх вопрос о памяти для хранения текстур далеко не праздный. Кроме того, память требуется не только для хранения текстур, но и для каркасов объектов.

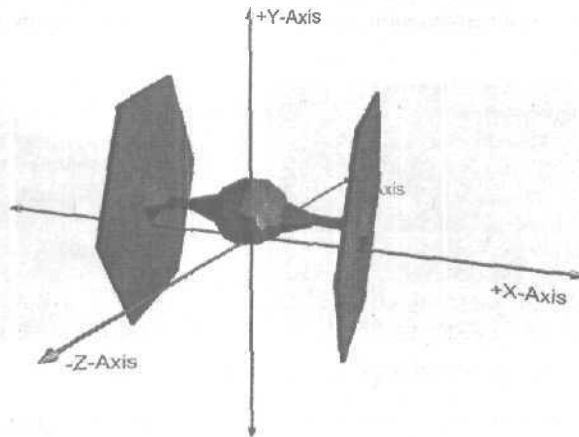


Рис. 6.11. Простая модель с небольшим количеством многоугольников

Например, если вы решите использовать предварительно сканированные анимированные каркасы, а не данные движения, то может потребоваться весьма большое количество памяти. Допустим, вы работаете с объектом, изображенным на рис. 6.11, который состоит примерно из 200–300 многоугольников и 200 вершин. Выполним небольшие расчеты количества памяти, необходимой для хранения вершин с поддержкой координат вершин, текстур и освещения. Допустим, для хранения данных используется простейшая структура данных.

```
typedef VERTEX3DptL_TYP
{
    float x,y,z; // Позиция
    float u,v;   // Координаты текстуры
    float l;     // Значение освещения
} VERTEX3DptL, * VERTEX3DptL_PTR;
```

Это дает нам грубую оценку на 32-битовой платформе, как минимум,  $6 * \text{sizeof}(\text{float}) = 24$  байта на одну вершину. Наш объект состоит из 200 вершин, так что нам потребуется в 200 раз больше памяти, т.е. 4800 байт памяти для одного каркаса.

Однако мы даже не пытались учесть такие вещи, как разбиение многоугольников на треугольники, их цвета и т.п. Забудем обо всем этом на минутку и продолжим рассмотрение упрощенного примера. Пусть у нас для каждого движения/действия используется по 32 кадра анимации, а возможных движений — всего 6:

- ходьба;
- бег;
- прыжок;
- плавание;
- гибель;
- стрельба.

Итого 6 движений по 32 кадра, всего — 192 кадра. Но, как мы уже выяснили, каждый кадр — это 4800 байтов. Итого для объекта нам требуется  $192 \cdot 4800 = 921600$  байтов памяти. Если у нас имеется, скажем, 32 различных персонажа, значит, нам надо уже 29491200 байтов. Итого — больше 28 Мбайт памяти! Как видите, в плане памяти трехмерная игра — очень емкое в прямом смысле этого слова понятие... Главный вывод из приведенного подсчета в том, что при разработке трехмерной игры нам совершенно необходимы эффективные структуры данных, базы данных и загрузка и выгрузка данных по требованию. Конечно, диспетчер виртуальной памяти Windows помогает решить массу проблем, но все равно вы не можете загрузить в память всю необходимую информацию и работать с ней там — вы неизбежно столкнетесь с нехваткой памяти.

**СОВЕТ**

Кроме прочего, вы не должны забывать и о вопросах полосы пропускания. При работе с картами текстур и большими сетками реализация обычного перемещения "съедает" большой кусок полосы пропускания системной шины. Это означает, что важную роль приобретает упорядочивание данных и способ работы с ними. В этой ситуации окажется недопустимой роскошью загрузка объекта А, работа с ним, загрузка и работа с объектом В и повторная загрузка объекта А. Вы должны так упорядочивать данные, чтобы выполнить все, что требуется, с загруженными данными, прежде чем загрузить другие данные.

В заключение скажу, что трехмерную игру (или просто игру — в принципе ее размерность не имеет значения) можно считать простым потоком байтов, однако очень важную роль играет представление этого потока. Следовательно, о подходящих структурах данных и плане работы с информацией — вот о чем надо обязательно подумать, прежде чем приступить к кодированию. Справедливости ради следует сказать, что это относится практически к любой задаче программирования.

## Трехмерные системы координат

Пришло время поговорить о некоторых специфичных для трехмерных игр вещах. Начнем мы этот разговор с трехмерных систем координат.

Хотя мы уже знакомы с декартовыми, цилиндрическими и сферическими системами координат, мы еще не рассматривали, как трехмерные графические процессоры представляют реальные трехмерные объекты на разных этапах работы с ними.

Вообще говоря, в зависимости от того, как выполняется преобразование объекта до его вывода на экран, могут быть использованы различные системы координат. О них обычно говорят как о конвейерах визуализации (viewing pipeline). Это:

- координаты модели;
- мировые координаты;
- координаты камеры;
- координаты перспективы;
- экранные координаты.

Отметим, что здесь перечислены лишь основные классы координат. Впрочем, в дальнейшем рассмотрении мы ограничимся только приведенным списком.

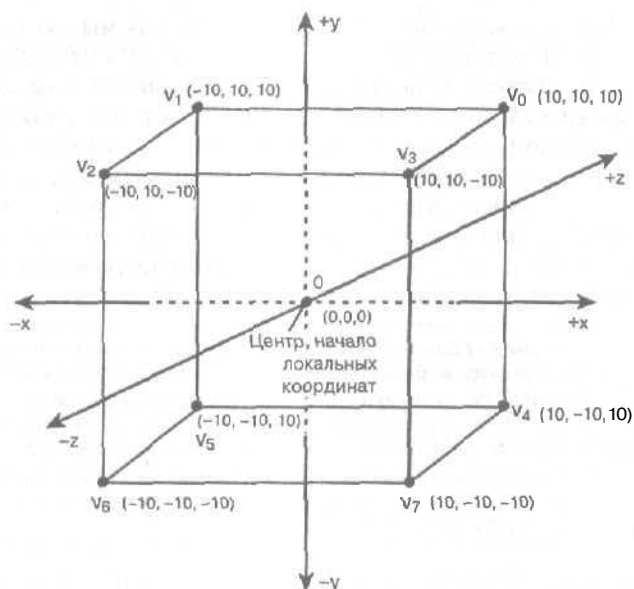


Рис. 6.12. Базовый куб в локальных координатах

## Координаты модели (локальные координаты)

Координаты модели (известные также как *локальные координаты*) представляют собой координаты трехмерного объекта в своей собственной локальной системе координат. Всякий раз, когда вы создаете трехмерный объект игры, вы обычно создаете и набор его собственных локальных осей координат, в которых **центр** объекта (или то, что вы хотите считать таковым) имеет координаты  $(0,0,0)$ . На рис. 6.12 показан куб, определенный в левой системе координат. Длина стороны данного куба равна 20, а центр куба находится в начале координат  $(0,0,0)$ . Вот вершины (точки), которые образуют куб.

Вершина 0:  $(10, 10, 10)$

Вершина 1:  $(-10, 10, 10)$

Вершина 2:  $(-10, 10, -10)$

Вершина 3:  $(10, 10, -10)$

Вершина 4:  $(10, -10, 10)$

Вершина 5:  $(-10, -10, 10)$

Вершина 6:  $(-10, -10, -10)$

Вершина 7:  $(10, -10, -10)$

Основная идея состоит в том, что в некоторый момент работы конвейера в трехмерной игре выполняется перенос модели в ее реальное положение в трехмерном мире.

Начало локальной системы координат  $(0,0,0)$  вашего объекта необязательно должно находиться в геометрическом центре объекта. **Предположим**, например, что рука робота имитируется при **помощи** модели, показанной на рис. 6.13. Каждое звено можно определить как сетку многоугольников со своими локальными центрами координат. При переносе объекта в реальный мир достаточно перенести начало локальной системы координат в конкретную точку и выполнить простейшие преобразования. Однако если звенья руки могут **вращаться** относительно друг Друга, более естественным будет использование

локальной системы координат, привязанной не к центрам, а к концам звеньев (рис. 6.13) — при этом одна и та же точка  $(0,0,0)$  в локальной системе координат будет как началом звена, так и точкой, вокруг которой выполняется его вращение, что существенно упрощает все необходимые математические вычисления.

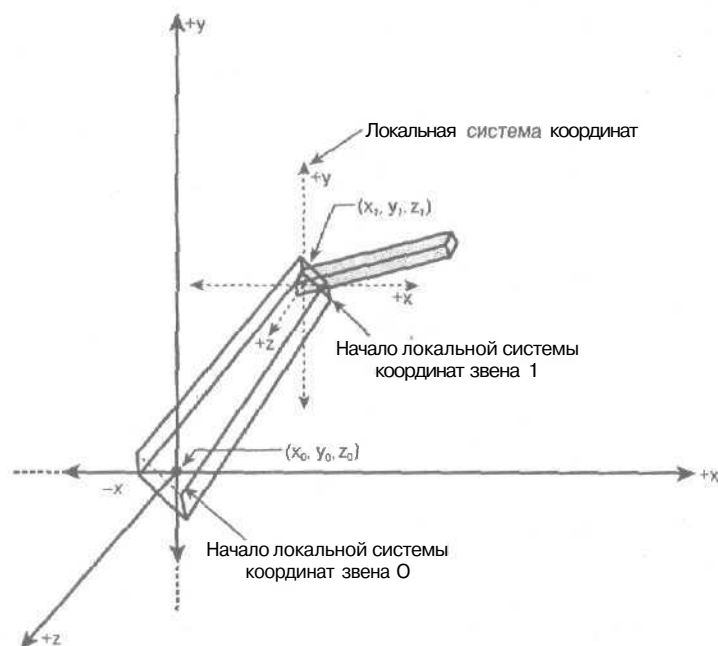


Рис. 6.13. Сочлененная рука робота и системы локальных координат

## Вопросы ориентации

Говоря о локальных координатах, я хотел бы упомянуть и о вопросах ориентации. Под *ориентацией* я подразумеваю то, что у вас должны быть некоторые соглашения о том, как определяются и загружаются ваши трехмерные объекты; в противном случае вы просто не будете знать, где у объектов верх, и где — перед. Предположим, например, что вы определили два объекта для трехмерной игры, показанные на рис. 6.14. Космический корабль имеет локальные координаты, построенные таким образом, что передняя часть корабля ориентирована вдоль отрицательного направления оси  $z$ , а верх корабля — вдоль положительного направления оси  $y$ . Робот же создан так, что его передняя часть ориентирована в положительном направлении оси  $z$ . Надеюсь, теперь вам понятно, в чем заключается проблема?

Есть два способа решения указанной проблемы. Первый метод состоит в определении соглашения, действующего при моделировании, согласно которому все модели должны быть ориентированы своей передней частью в положительном направлении оси  $z$ , а верхней частью — в положительном направлении оси  $y$  (или другое соглашение, но такое, которое четко увязывает локальную систему координат объекта с его ориентацией). Вторым методом заключается в моделировании объектов произвольным образом, но при этом должно предусматриваться программное обеспечение, которое позволяет вычислить главную ось каждого объекта (ось, вдоль которой объект имеет наибольшую длину), а также вторую главную ось, и согласовывать их с положительным направлением оси  $y$

(или с любой другой; главное, чтобы это правило действовало в отношении всех объектов). При этом, как бы ни был ориентирован загруженный объект, мы всегда можем определить его ориентацию в пространстве. Мой совет — все же использовать первый способ, поскольку, каким бы интеллектуальным не было программное решение, оно опирается исключительно на геометрические характеристики объекта и легко может принять неверное решение о его ориентации.

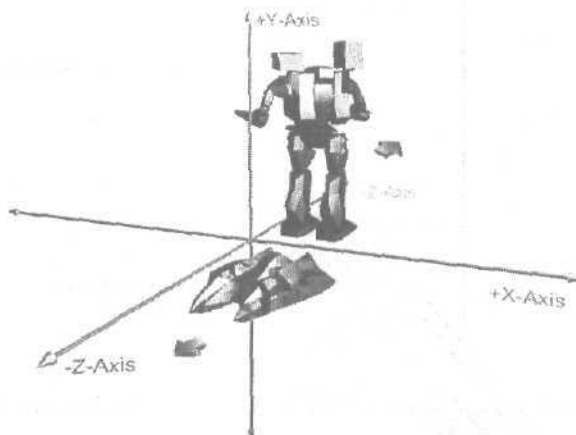


Рис. 6.14. Ориентация загруженных моделей

#### СОВЕТ

Даже при использовании соглашений времени моделирования, вы можете захотеть поворачивать, масштабировать или перемещать объект в процессе загрузки. Это вполне возможно, и вы можете разработать свои функции, которые будут трансформировать объект в процессе загрузки. В этом случае соответствующие изменения будут претерпевать и локальная система координат, передаваемая системе.

### Вопросы масштабирования

Предположим, одна модель, представляющая собой автомобиль, имеет радиус 100, и другая модель (небольшой велосипед) тоже имеет тот же размер (рис. 6.15). Понятно, что это лишено смысла, так как велосипед размером с автомобиль — нонсенс. Решение данной проблемы состоит в том, чтобы создавать все ваши модели в одном масштабе, в котором единица соответствует какому-то фиксированному значению длины в реальном мире — например, сантиметру. Другое решение — моделировать все объекты в некотором нормализованном локальном мире 1x1x1, а затем, при загрузке, масштабировать их. Лично я предпочитаю создавать объекты в одном масштабе. Заметим, что это не мешает функции загрузки все же выполнять масштабирование (возможно, разной величины вдоль разных осей) наряду с поворотом объектов. Это может оказаться весьма полезным — например, в случае с астероидами. Можно создать одну модель астероида, которую затем по-разному масштабировать и поворачивать, создавая иллюзию различных астероидов, что существенно упрощает вашу работу по моделированию трехмерных объектов.

### Левые и правые системы координат

В трехмерной игре при определении каждого объекта вы можете использовать либо левую, либо правую систему координат. Но, конечно, вы должны быть последовательны. Я предпочитаю левую систему координат и положительное направление оси  $z$  внутрь экрана. Однако ничто не мешает вам использовать в работе правую систему координат —

вся разница будет *лишь* в направлении оси z, отражение которой для перехода в левую систему координат делается очень просто. В принципе везде можно обойтись правой системой координат, но тогда вы будете должны сами разобраться, в чем ваши программы и формулы должны отличаться от моих.

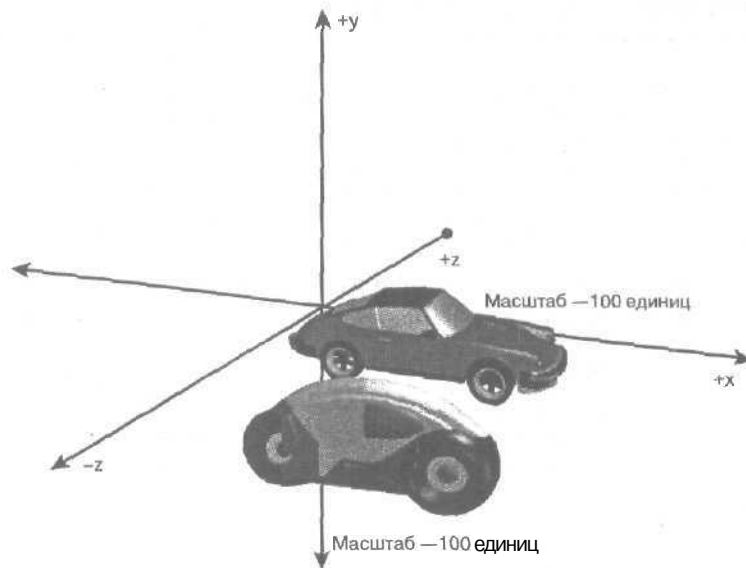


Рис. 6.15. Различия в масштабах должны быть нормализованы

## Мировые координаты

После того как вы создали все необходимые трехмерные модели, их надо разместить в нашем игровом мире. На этом этапе в игру вступают *мировые координаты*. Это реальные координаты виртуального игрового мира, в котором перемешаются и изменяются наши объекты. Первый вопрос, связанный с мировыми координатами — какими они должны быть? Другими словами, насколько велик должен быть мир игры? Должен ли он иметь размеры 1x1x1, 10x10x10 или 1 000 000x1 000 000x1 000 000?

Например, если использовать соглашение о том, что *единица* длины в виртуальном мире соответствует метру в реальном, и вы хотите, чтобы размеры вашего мира были порядка 100x100x100 km, то это означает, что сетка вашей игры имеет следующую размерность;

$$-50\,000 \leq x \leq 50\,000$$

$$-50\,000 \leq y \leq 50\,000$$

$$-50\,000 \leq z \leq 50\,000$$

Это вполне разумные величины, легко обрабатываемые при помощи типа данных float. С другой стороны, если вы хотите, чтобы единица координат модели соответствовала 1 cm, а игровой мир представлял собой куб размером 1 000 000 km, то диапазоны значений координат становятся довольно неприятными:

$$-0.5 \times 10^{10} \leq x \leq 0.5 \times 10^{10}$$

$$-0.5 \times 10^{10} \leq y \leq 0.5 \times 10^{10}$$

$$-0.5 \times 10^{10} \leq z \leq 0.5 \times 10^{10}$$

Эти значения превышают возможности представления целых чисел 32-битовыми целыми (диапазон значений которых приближенно от  $-2 \times 10^9$  до  $2 \times 10^9$ ), и для работы с координатами в таком случае требуются числа с плавающей точкой. Словом, получившиеся координаты слишком велики. При выборе размеров игрового мира можно придерживаться приближенного правила, в соответствии с которым вы должны иметь возможность представить границы мира, как минимум, с точностью 3-4 знака после десятичной точки.

НА ЗАМЕТКУ

С вопросами относительно размера игрового мира тесно связаны вопросы скорости перемещения объектов. Например, однажды я делал игру-«стрелялку» с размером игрового мира  $4096 \times 4096 \times 4096$ . Из конца в конец этого мира игрок мог добраться примерно за 15 секунд. Это вполне приемлемое время для игры такого рода, но для космической игры, очевидно, требуется существенно больший размер — хотя бы в силу большей относительной скорости космических кораблей.

## Размещение объектов в мировых координатах

Рассмотрим теперь, каким образом следует преобразовывать объект из локальных координат в мировые.

Вспомним, что цель локальных координат — определение объекта, но затем этот объект должен быть размещен в пространстве игры и получить окончательные значения мировых координат. В качестве примера рассмотрим еще раз рис. 6.12. Здесь вы видите простой трехмерный куб, определенный в локальных координатах.

Вершина 0: ( 10, 10, 10)  
Вершина 1: (-10, 10, 10)  
Вершина 2: (-10, 10, -10)  
Вершина 3: ( 10, 10, -10)  
Вершина 4: ( 10, -10, 10)  
Вершина 5: (-10, -10, 10)  
Вершина 6: (-10, -10, -10)  
Вершина 7: ( 10, -10, -10)

Мы определяем данный объект, используя структур данных POINT3D.

```
POINT3D cube_model[8] = {  
    { 10, 10, 10}, // Вершина 0  
    {-10, 10, 10}, // Вершина 1  
    {-10, 10, -10}, // Вершина 2  
    { 10, 10, -10}, // Вершина 3  
    { 10, -10, 10}, // Вершина 4  
    {-10, -10, 10}, // Вершина 5  
    {-10, -10, -10}, // Вершина 6  
    { 10, -10, -10}, // Вершина 7  
};
```

Конечно, мы можем определить все стороны куба как многоугольники, состоящие из 4 вершин.

Сторона 0: (0,1,2,3)  
Сторона 1: (4,7,6,5)  
Сторона 2: (0,3,7,4)  
Сторона 3: (2,3,7,6)  
Сторона 4: (1,5,6,2)  
Сторона 5: (0,4,5,1)

Однако в действительности для преобразования локальных координат в мировые определение многоугольников нам не потребуется.

Итак, у нас есть список вершин объекта в локальных координатах и мы хотим преобразовать его в мировые координаты. Для этого нам нужно знать, где в мировых координатах будет расположен *центр* объекта. Предположим, что наш мир имеет размер  $1000 \times 1000 \times 1000$  и мы хотим поместить объект в точке  $(world\_x, world\_y, world\_z)$ , как показано на рис. 6.16. Это легко выполнить при помощи следующего простейшего кода.

```
POINT3D cube_world[8]; // Мировые координаты куба
```

```
// Преобразуем все вершины куба
for (int vertex = 0; vertex < 8; vertex++)
{
    cube_world[vertex].x = cube_model[vertex].x + world_x;
    cube_world[vertex].y = cube_model[vertex].y + world_y;
    cube_world[vertex].z = cube_model[vertex].z + world_z;
} // for
```

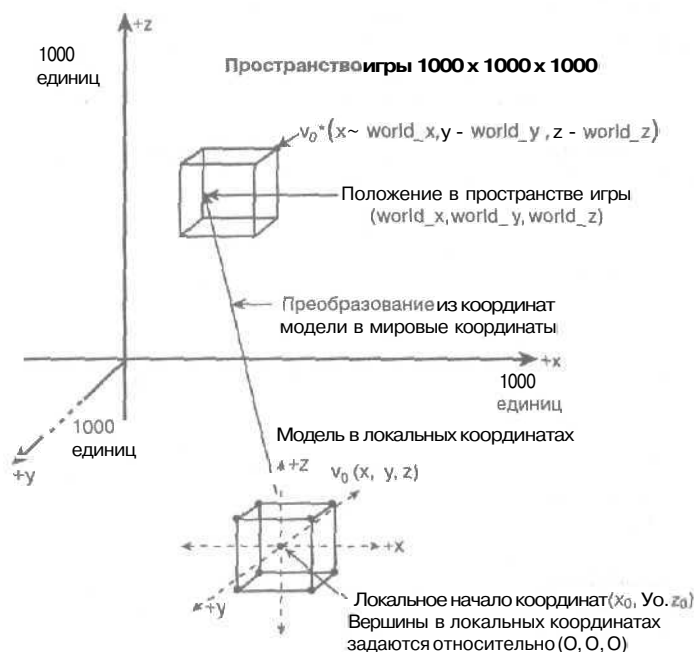


Рис. 6.16. Размещение объекта в пространстве игры

Обратите внимание, что мы *не* изменяем локальные координаты куба — это очень важно! Если бы мы поступили таким образом, модель куба была бы для нас потеряна. Поэтому мы сохраняем преобразованные координаты в новом списке вершин. Это может показаться напрасной тратой памяти, но этого невозможно избежать. Действительно, в процессе перемещения по конвейеру визуализации может быть задействовано несколько различных версий координат. Мы постараемся свести количество копий к минимуму, но обычно меньше чем двумя-тремя копиями координат обойтись не удастся...

В любом случае приведенный алгоритм — не более чем обычный перенос, а потому преобразование локальных координат в мировые можно представить при помощи стандартной матричной операции. Обозначим матрицу преобразования из координат модели в мировые  $T_{mw}$ . Вспомним, что для поддержки переноса при помощи матричной опера-

ции нам нужны четырехмерные однородные координаты, и матрица будет выглядеть следующим образом:

$$T_{mw} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \text{world\_x} & \text{world\_y} & \text{world\_z} & 1 \end{bmatrix}$$

НА ЗАМЕТКУ

Я очень часто именую матрицу как  $T$  с нижними индексами, наподобие  $T_{mw}$ . Здесь  $T$  обозначает обобщенное преобразование (transformation). Но когда мы будем говорить о переносах и поворотах (translation и rotation),  $T$  будет обозначать матрицу переноса, а  $R$  — матрицу поворота. Иными словами, когда вы видите матрицу  $T$  — это либо матрица обобщенного преобразования, либо переноса.

Если помните, у нас есть масса "интеллектуальных" подпрограмм перемножения матриц, которые знают, как умножить трехмерный вектор на матрицу  $4 \times 3$  и/или четырехмерный вектор на матрицы  $4 \times 3$  и  $4 \times 4$  и т.д. Главное в том, что для реализации переноса при помощи матричного перемножения нам требуется четвертая строка. Исходя из этого, преобразование локальных координат в мировые можно записать следующим образом.

#### Уравнение 6.1. Преобразование локальных координат в мировые

$$\begin{aligned} [x_m \ y_m \ z_m \ 1] \cdot T_{mw} &= \\ &= [x_m \ y_m \ z_m \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \text{world\_x} & \text{world\_y} & \text{world\_z} & 1 \end{bmatrix} = \\ &= [x_m + \text{world\_x} \ y_m + \text{world\_y} \ z_m + \text{world\_z} \ 1] = \\ &= [x_w \ y_w \ z_w \ 1]. \end{aligned}$$

Само собой разумеется, в этом примере мы используем четырехмерные однородные координаты для упрощения математики. Вполне можно использовать трехмерные координаты, если предположить наличие фантомного четвертого элемента  $w$ , равного 1.0, так что умножение матриц  $4 \times 3$  или  $4 \times 4$  имеет определенный смысл. И наконец, для получения трехмерных координат из однородных четырехмерных требуется разделить  $[x_w \ y_w \ z_w]$  на множитель  $w$  (однако, поскольку  $w = 1.0$ , мы можем обойтись и без деления).

В заключение хочу сказать, что преобразование локальных координат в мировые — тривиальная операция: надо просто перенести центр каждого объекта в требуемую позицию в мировых координатах —  $(\text{world\_x}, \text{world\_y}, \text{world\_z})$ . Это, в свою очередь, выполняется переносом всех вершин модели и сохранения полученных результатов во вторичной структуре данных. Все необходимые для масштабирования и повороты объекта лучше выполнять в локальных координатах, а затем переносить объект в реальный мир. На рис. 6.17 показана рассмотренная нами к этому моменту часть конвейера визуализации.

## Координаты камеры

До сих пор все было очень просто. Как локальные, так и мировые координаты использовались для определения трехмерных объектов в пространстве. Локальные координаты определяли один объект или их набор в их собственном трехмерном пространстве и

системе координат. Мировые координаты определяют виртуальный трехмерный мир в абсолютных величинах. Однако не следует забывать о том, что цель трехмерной игры — в том, чтобы в нее можно было играть, а значит, чтобы объекты могли двигаться вокруг трехмерной камеры. Следовательно, нам необходим некоторый способ увидеть трехмерные объекты. Для этого нам нужны *координаты камеры* и *преобразование мировых координат в координаты камеры*.

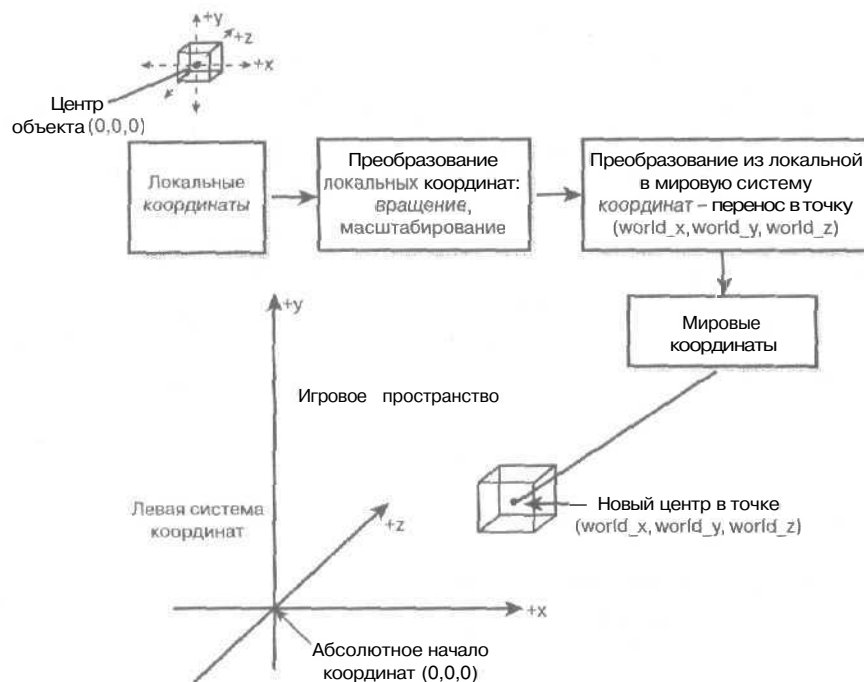


Рис. 6.17. Конвейер преобразования локальных координат в мировые

## Область обзора

Для того, чтобы увидеть сцену в трехмерном мире, обычно используется метод, состоящий в размещении виртуальной камеры в некоторой позиции игрового мира и затем обзор мира с определенным полем зрения (рис. 6.18). Как видите, у нас имеется трехмерная мировая система координат с виртуальной камерой, расположенной в точке  $(cam\_x, cam\_y, cam\_z)$ , направление которой определяется углами  $(ang\_x, ang\_y, ang\_z)$ , которые известны также как крен, рыскание и тангаж. Кроме того, как видите, есть определенная область пространства, которое "видит" камера. Ее можно определить с помощью параметров *FOV* (field of view/, поле обзора) в горизонтальном и вертикальном направлении, которые обычно имеют значения в диапазоне  $60\text{--}130^\circ$ . Кроме того, имеются ближняя и дальняя плоскости отсечения, отделяющие область, в которой находятся видимые объекты (объекты за пределами определяемой этими плоскостями области пространства невидимы для камеры). И наконец, есть плоскость проекции, которая представляет собой математическую плоскость, на которую проецируется трехмерное изображение, так что мы получаем возможность представить трехмерное пространство на плоском экране компьютера. Теперь поговорим более подробно о шести стенках, которые образуют наш "ящик" обзора.

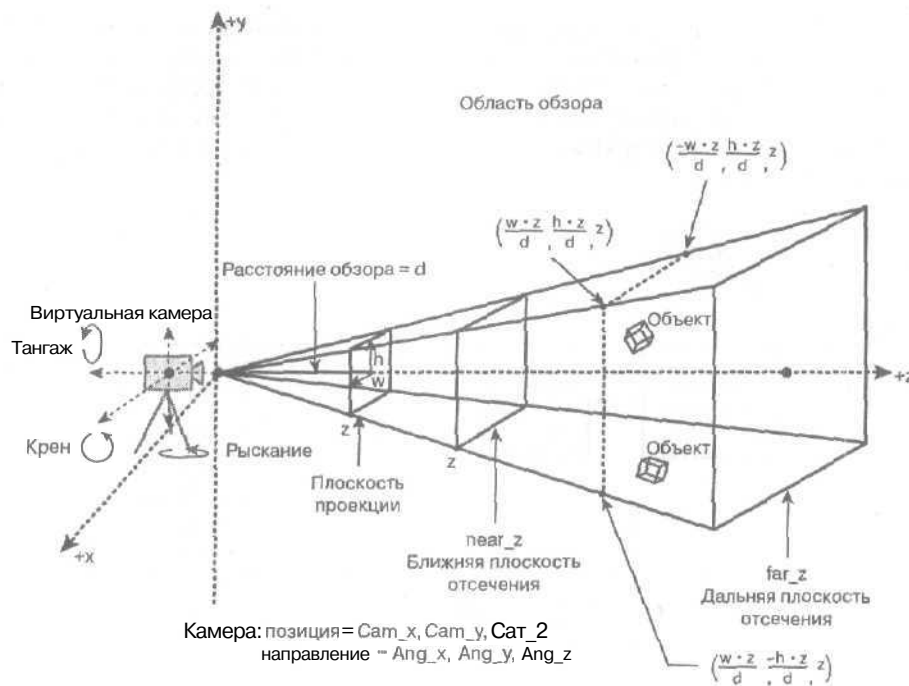


Рис. 6.18. Трехмерная область обзора

### Ближняя и дальняя плоскости отсечения

Эти плоскости имеют координаты по оси  $z$ , соответственно,  $near\_z$  и  $far\_z$ , и уравнения этих плоскостей выглядят просто как  $z = near\_z$  и  $z = far\_z$  или, более обобщенно:

ближняя плоскость:  $0 \cdot x + 0 \cdot y + 1 \cdot z = near\_z$

дальняя плоскость:  $0 \cdot x + 0 \cdot y + 1 \cdot z = far\_z$

Эти плоскости перпендикулярны направлению обзора и представляют собой геометрические пределы области, которая может быть визуализирована. Другими словами, объекты, расположенные ближе ближней плоскости и дальше дальней плоскости отсечения, не будут визуализированы. Слишком удаленные объекты становятся для камеры просто точками, которые все равно невозможно рассмотреть, а слишком близкие невозможно рассмотреть из-за их близости — как, например, вы не можете рассмотреть сетку непосредственно перед вашими глазами.

### Стены области обзора

Область обзора представляет собой усеченную пирамиду, определяемую ближней и дальней плоскостями отсечения, и боковыми плоскостями. В нашем случае мы используем угол обзора  $90^\circ$ , так что каждая из этих плоскостей расположена под углом  $45^\circ$  к направлению обзора, как показано на рис. 6.19. Таким образом, данные плоскости описываются простыми уравнениями  $|x| = z$  и  $|y| = z$ . Данные плоскости используются для отбора визуализируемых объектов так же, как и плоскости отсечения. Объекты, находящиеся за пределами этих плоскостей должны быть отсечены или полностью удалены из конвейера.

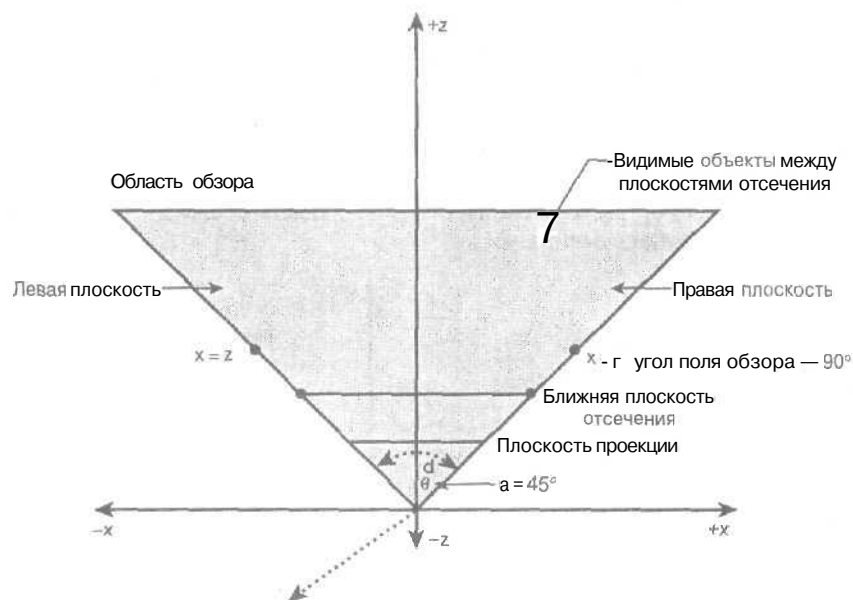


Рис. 6.19. Схематичный вид сверху на область обзора

### Связь координат камеры и мировых координат

Проблема состоит в следующем: мы хотим преобразовать объекты в трехмерном мире, представив их относительно камеры. Другими словами, мы хотим иметь возможность связать мировые координаты объектов с камерой как можно более простым способом. Эта идея и лежит в основе преобразования мировых координат в координаты камеры.

Взгляните на рис. 6.19, где камера расположена в точке  $(0,0,0)$  мировых координат, направлена вдоль оси  $z$ , с углами обзора  $90^\circ$  (равенство  $FOV=90^\circ$  не является обязательным, просто оно существенно упрощает расчеты благодаря тривиальности уравнений плоскостей в данном случае). Если в области обзора находится какой-либо объект — он должен быть визуализирован, в противном случае он должен быть невидим. А дело вот в чем — в 99.99999% случаев камера не находится в начале координат и не направлена вдоль оси  $z$ . Обычно камера по сути присоединена к лицу игрока.

Таким образом, главный вопрос состоит в следующем: как переместить мир, чтобы камера оказалась расположена в точке  $(0,0,0)$  и была ориентирована вдоль оси  $z$ , причем чтобы "верх" располагался в положительном направлении оси  $z$  (т.е. чтобы углы, определяющие направление камеры, также имели значения  $(0,0,0)$ ). Это и есть искомое преобразование, и оно оказывается на удивление не сложным.

### Преобразование мировых координат в координаты камеры

Это преобразование, которое приводит к размещению камеры в точке  $(0,0,0)$  с углами направления  $(0,0,0)$ , выполняется в два этапа — переноса и поворота.

#### Перенос

Посмотрите на рис. 6.20а. На нем представлен упрощенный трехмерный мир с одним объектом и камерой в некоторой позиции  $(cam\_x, cam\_y, cam\_z)$ . Для просто-

ты мы считаем, что углы направления камеры —  $(0,0,0)$ . Теперь, если мы переместим камеру в точку  $(0,0,0)$ , то что случится с объектом, расположенным в точке  $(world\_x, world\_y, world\_z)$ , если он останется в том же положении относительно камеры, что и раньше? Если обратиться к рис. 6.20б, то становится понятно, что для сохранения относительного положения камеры и объекта из координат последнего следует вычесть координаты камеры до переноса. Все относительные расстояния и углы при этом **остаются** неизменными.

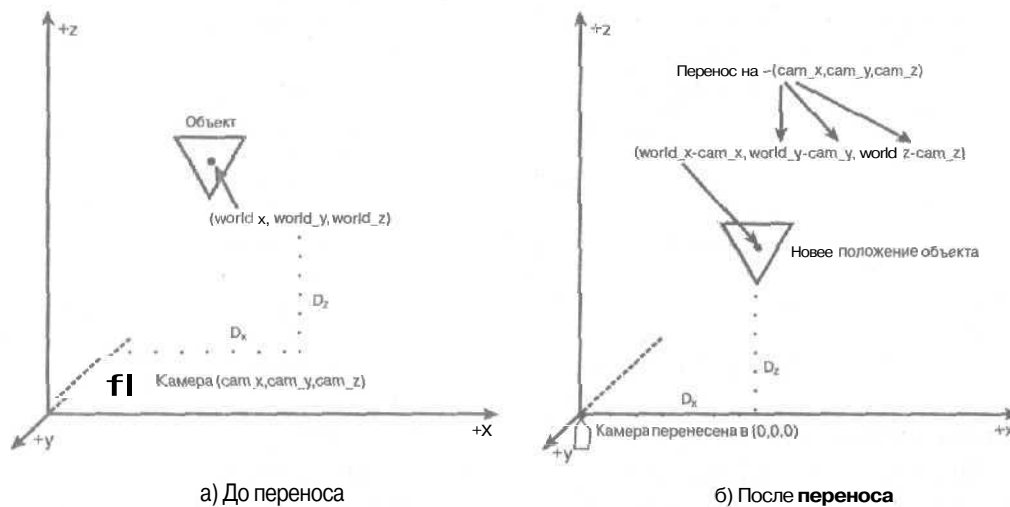


Рис. 6.20. Преобразование координат: перенос

Итак, первый этап преобразования мировых координат в координаты камеры сводится к вычитанию из координат всех объектов мира (в том числе и камеры) координат последней в игровом мире. Соответствующее преобразование для уже рассматривавшегося нами куба представлено ниже.

**POINT3D cube\_camera[8]; // Координаты относительно камеры**

```
// Перенос каждой вершины на (-cam_x, -cam_y, -cam_z)
for (int vertex = 0; vertex < 8; vertex++)
{
    cube_camera[vertex].x = cube_model[vertex].x - cam_x;
    cube_camera[vertex].y = cube_model[vertex].y - cam_y;
    cube_camera[vertex].z = cube_model[vertex].z - cam_z;
} // for
```

Как видите, нам опять необходим новый набор вершин для хранения координат объекта в координатах камеры. Кроме того, видно, что это преобразование координат аналогично преобразованию локальных координат в **мировые**, так что можно подумать над объединением этих двух шагов в один. И хотя мы рассмотрели только половину преобразования мировых координат в координаты камеры, я должен сказать, что вполне можно объединить **многие** преобразования из конвейера визуализации при **помощи** конкатенации **матриц**. Впрочем, часто имеет смысл делать это **последовательно**, так как, например, нам может не потребоваться приводить все объекты к координатам камеры. При работе

конвейера многие объекты и/или многоугольники могут оказаться ненужными из-за отсечения, удаления скрытых поверхностей и т.п. Тем не менее, вполне может быть, что для преобразования локальных координат в экранные потребуются единственная матрица.

Данное преобразование, как и другие, можно записать в матричной форме. Полагая, что матрица преобразования положения камеры —  $T_{cam}$ , обратная матрица  $T_{cam}^{-1}$  находится простым изменением знака множителей переноса:

$$T_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}.$$



Вспомните, что обращение матрицы переноса осуществляется путем изменения знака множителей переноса в нижней строке матрицы. Обращение матрицы поворота осуществляется путем изменения знака угла каждого синуса и косинуса в матрице, или простым транспонированием матрицы поворота.

Таким образом, если мы умножим каждую вершину в мировых координатах на матрицу  $T_{cam}^{-1}$ , то все они будут корректно перенесены в систему координат, связанную с камерой (в которой камера расположена в точке  $(0,0,0)$ ). Вот как математически выглядит первый этап преобразования мировых координат в координаты камеры.

#### Уравнение 6.2. Этап переноса преобразования мировых координат в координаты камеры

$$\begin{aligned} [x_w \ y_w \ z_w \ 1] \cdot T_{cam}^{-1} &= \\ &= [x_w \ y_w \ z_w \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix} = \\ &= [x_w - cam\_x \ y_w - cam\_y \ z_w - cam\_z \ 1]. \end{aligned}$$

Само собой разумеется, что при реализации данного переноса в коде можно воспользоваться умножением матриц.

```
POINT3D cube_camera[8]; // Координаты относительно камеры
```

```
// Определение матрицы переноса
MATRIX4X4 Tcam_inv = { {1, 0, 0, 0},
                        {0, 1, 0, 0},
                        {0, 0, 1, 0},
                        {-cam_x, -cam_y, -cam_z, 1} };
```

```
// Преобразование каждой вершины путем умножения на матрицу
for (int vertex = 0; vertex < 8; vertex++)
{
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
                        &Tcam_inv,
                        cube_camera[vertex]);
} // for
```

В настоящий момент промежуточные значения координат вершин находятся в массиве `cube_camera[]`. При выполнении матричного умножения вам следует убедиться, что исходные и получаемые в результате расчета координаты хранятся в разных местах; в противном случае результат может оказаться некорректным из-за использования при вычислениях промежуточных значений. Ни в коем случае при работе с матрицами нельзя передавать в функцию один и тот же массив в качестве как исходного, так и массива для получения возвращаемых значений.

Теперь приступим ко второму этапу преобразования мировых координат в координаты камеры.

### Поворот

На рис. 6.21а показан по сути тот же пример, что и на рис. 6.20, однако в этот раз камера, расположенная в точке с координатами  $(cam\_x, cam\_y, cam\_z)$ , имеет направление, определяемое углами  $(0, ang\_y, 0)$ . Другими словами, камера имеет ненулевой угол рыскания, и обзор направлен не в положительном направлении оси  $z$ . Если мы определим, как надо повернуть камеру для того, чтобы она была направлена вдоль оси  $z$ , задачу можно считать решенной.

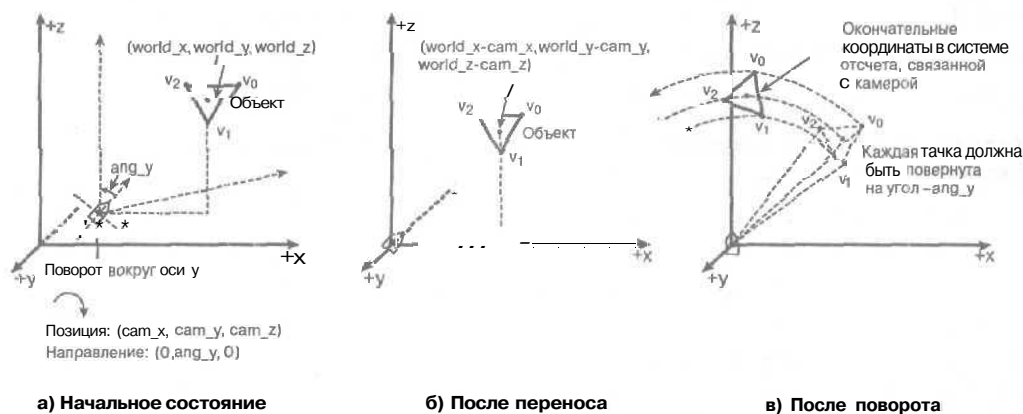


Рис. 6.21. Преобразование координат: поворот

Первое, что мы делаем, — это преобразование переноса мировых координат всех объектов с тем, чтобы камера находилась в точке  $(0,0,0)$ . Это выполняется путем переноса всех объектов, включая камеру, на отрицательный радиус-вектор камеры  $(-cam\_x, -cam\_y, -cam\_z)$ , что представляет собой не что иное, как умножение каждой точки на матрицу  $T_{cam}^{-1}$ .

После этого мы получаем ситуацию, показанную на рис. 6.21б. Камера расположена в точке  $(0,0,0)$ ; относительное положение объекта остается прежним, однако сама камера имеет направление, определяемое углами  $(0, ang\_y, 0)$ . Все, что мы должны сделать для того, чтобы камера была направлена вдоль оси  $z$  — это повернуть ее вокруг оси  $y$  на угол  $-ang\_y$ , что представляет собой инверсию поворота камеры в начальном направлении. Однако мы должны повернуть не только камеру, но и все точки игрового пространства, чтобы их положение относительно камеры осталось неизменным. Результат такого поворота показан на рис. 6.21в.

На рисунке видно, что мы достигли желаемого положения камеры в точке  $(0,0,0)$  и направления вдоль оси  $z$  (т.е. описываемого углами  $(0,0,0)$ ). Положение всех объектов относительно камеры и друг относительно друга остается при этом прежним. Что действительно изменилось — это абсолютные координаты вершин. Это и есть полное преобразование мировых координат в координаты камеры, расположенной в точке  $(cam\_x, cam\_y, cam\_z)$  и ориентированной в направлении, определяемом углами Эйлера  $(0, ang\_y, 0)$ .

Но что будет в общем случае произвольных углов  $(ang\_x, ang\_y, ang\_z)$ ? Преобразование будет таким же: сначала выполняется перенос при помощи матрицы  $T_{cam}^{-1}$ , а затем — серия поворотов. Поскольку ориентация камеры определяется тремя углами, есть  $3! = 6$  способов выполнить последовательность поворотов.

Последовательность 1:  $xzy$

Последовательность 2:  $xzy$

Последовательность 3:  $yxz$

Последовательность 4:  $yzx$

Последовательность 5:  $zxy$

Последовательность 6:  $zyx$

Большинство людей предпочитает третью или шестую последовательности. Это связано с естественным представлением, что первый поворот — поворот головы вправо-влево, а второй — вверх-вниз (последовательность  $yx$ ). Покачивание головой от плеча к плечу не так естественно для человека — это скорее собачья привычка. Впрочем, последовательность выполнения поворотов значения не имеет, однако для определенности мы будем использовать последовательность 3.

Итак, нам надо выполнить три поворота — вокруг оси  $y$ , который мы обозначим как  $R_{camy}^{-1}$ , затем вокруг оси  $x$  ( $R_{camx}^{-1}$ ) и вокруг оси  $z$  ( $R_{camz}^{-1}$ ). Данные повороты должны быть обратными к ориентации камеры. Таким образом, полная последовательность поворотов, которая должна быть выполнена для каждой вершины, выглядит следующим образом.

**Уравнение 6.3. Преобразование мировых координат в координаты камеры, расположенной в точке  $(cam\_x, cam\_y, cam\_z)$  и ориентированной в направлении  $(ang\_x, ang\_y, ang\_z)$**

$$T_{wc} = T_{cam}^{-1} \cdot R_{camy}^{-1} \cdot R_{camx}^{-1} \cdot R_{camz}^{-1}$$

Конечно, можно объединить все матрицы в одну матрицу преобразования мировых координат в координаты камеры  $T_{wc}$  и проводить все вычисления для вершин с ней:  $[x_w \ y_w \ z_w \ 1] \cdot T_{wc}$ . Правда, неплохо получилось?

#### СОВЕТ

Обратите внимание на то, что мы используем обратные матрицы переноса и поворота. Если рассматривать камеру как объект, то для ее ориентации и размещения в пространстве игры требуется выполнить последовательность преобразований

$$R_{camz} \cdot R_{camx} \cdot R_{camy} \cdot T_{cam}$$

Обратите внимание также на то, что этапы вращения и переноса выполняются в обратном порядке, а сами матрицы не инвертированы. Заметим, что по сути камера — не что иное, как единичный вектор, расположенный в точке  $(cam\_x, cam\_y, cam\_z)$  и ориентированный в направлении  $(ang\_x, ang\_y, ang\_z)$ .

На практике вам надо вычислить матрицу  $T_{wc} = T_{cam}^{-1} \cdot R_{camy}^{-1} \cdot R_{camx}^{-1} \cdot R_{camz}^{-1}$ , что можно сделать вручную, на бумаге и **использовать** полученные значения для инициализации матрицы, либо определить независимые **матрицы** и перемножить их для получения конечного результата. В любом случае вы получаете только одну матрицу преобразования, которую и следует применить ко всем точкам.

Однако при вычислении матрицы  $T_{wc}$  вручную можно **воспользоваться** алгебраическими преобразованиями для упрощения умножения матриц, но я сторонник того, что человек **должен** думать, а компьютер — считать, так что давайте заставим его поработать.

```
// Камера находится в точке (cam_x, cam_y, cam_z); ее
// ориентация — (ang_x, ang_y, ang_z). Последовательность
// поворотов — y*x*z
```

```
// Матрица переноса
MATRIX4X4 Tcam_inv = {
    {1, 0, 0, 0},
    {0, 1, 0, 0},
    {0, 0, 1, 0},
    {-cam_x, -cam_y, -cam_z, 1}
};
```

```
// Теперь нам надо определить матрицы поворота
// для осей x,y,z
```

```
// Определение обратной матрицы поворота вокруг оси x. Не
// забудьте, что мы подставляем в матрицу отрицательное
// значение угла для получения обратного поворота и
// инвертированной матрицы
MATRIX4X4 Rcamx_inv = {
    {1, 0, 0, 0},
    {0, cos(-ang_x), sin(-ang_x), 0},
    {0, -sin(-ang_x), cos(-ang_x), 0},
    {0, 0, 0, 1}
};
```

```
// Определение обратной матрицы поворота вокруг оси y. Не
// забудьте, что мы подставляем в матрицу отрицательное
// значение угла для получения обратного поворота и
// инвертированной матрицы
MATRIX4X4 Rcamy_inv = {
    {cos(-ang_y), 0, -sin(-ang_y), 0},
    {0, 1, 0, 0},
    {sin(-ang_y), 0, cos(-ang_y), 0},
    {0, 0, 0, 1}
};
```

```
// И наконец, определение обратной матрицы поворота вокруг
// оси z. Не забудьте, что мы подставляем в матрицу
// отрицательное значение угла для получения обратного
// поворота и инвертированной матрицы
MATRIX4X4 Rcamz_inv = {
```

```

    {cos(-ang_z), sin(-ang_z), 0, 0},
    {-sin(-ang_z), cos(-ang_z), 0, 0},
    {0, 0, 1, 0},
    {0, 0, 0, 1}
};

// Теперь мы готовы к проведению вычислений. Выполним
// произведение матриц
// Tcam_inv * Rcamy_inv * Rcamx_inv * Rcamz_inv

MATRIX4X4 Mtemp1, Mtemp2, Tcam;

Mat_Mul_4X4(&Tcam_inv, &Rcamy_inv, &Mtemp1);
Mat_Mul_4X4(&Rcamx_inv, &Rcamz_inv, &Mtemp2);

// Не забывайте о порядке умножения!
Mat_Mul_4X4(&Mtemp1, &Mtemp2, &Tcam);

// Теперь мы можем преобразовать все необходимые вершины из
// мировой системы координат в систему координат камеры
for (int vertex = 0; vertex < 8; vertex++)
{
    // Умножение точки cube_world[] на матрицу камеры
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
        &Tcam,
        cube_camera[vertex]);
} // for

```



Вы можете упростить определение матрицы, воспользовавшись тождествами  $\sin(-x) \equiv -\sin(x)$  и  $\cos(-x) \equiv \cos(x)$ .

Теперь в рассматриваемый нами конвейер визуализации можно добавить очередной компонент — преобразование мировых координат в координаты камеры (рис. 6.22). Однако следует заметить, что диаграмма не очень точна — в ней отсутствуют некоторые промежуточные этапы, наподобие отбраковки и отсечения объектов, но вскоре мы поговорим и об этом,

## Дополнительные вопросы

Один из последних вопросов, связанных с камерой, состоит в том, что преобразование мировых координат в координаты камеры зависит от представления виртуальной камеры. В этой вводной главе мы используем примитивную систему камеры, которая приводит ко множеству проблем, в частности к *блокировке подвески*.

Такая блокировка возникает при повороте вокруг одной оси до точки, когда камера оказывается ориентированной вдоль другой оси, и вращение вокруг этой оси не дает никакого перемещения оси обзора камеры. Взгляните на рис. 6.23. Если мы сначала повернем камеру относительно оси  $y$  на угол  $90^\circ$ , так что ее ось обзора совпадет с осью  $x$ , то дальнейший поворот камеры вокруг оси  $x$  по сути ничего не дает.

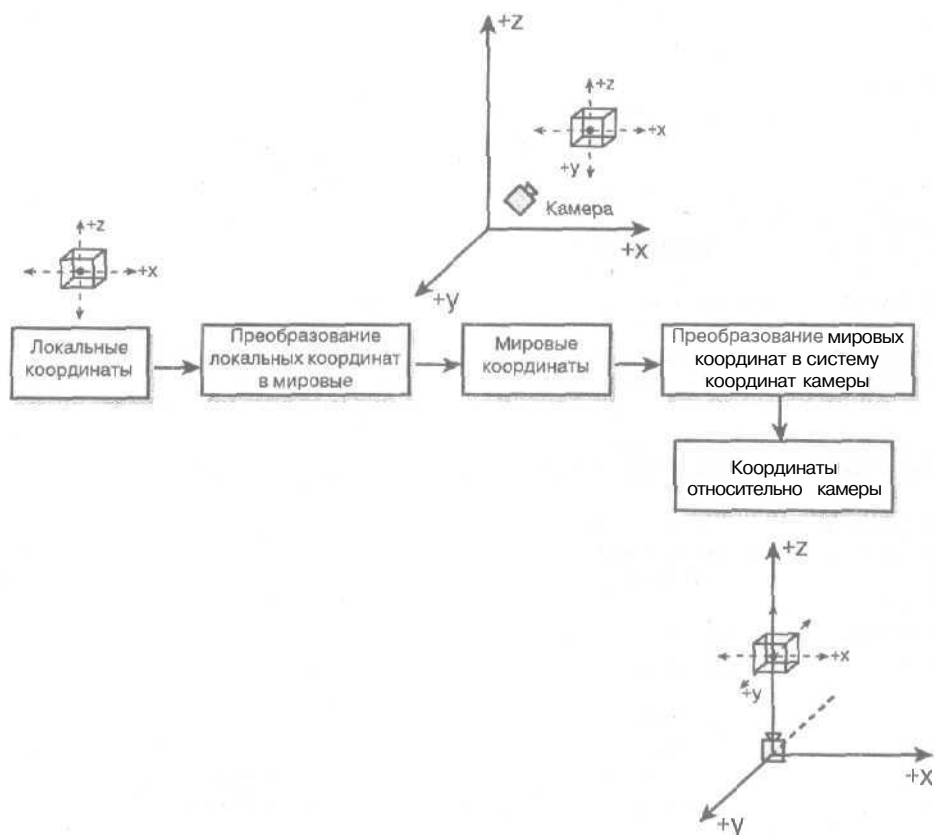


Рис. 6.22. Конвейер визуализации с преобразованием в систему координат камеры

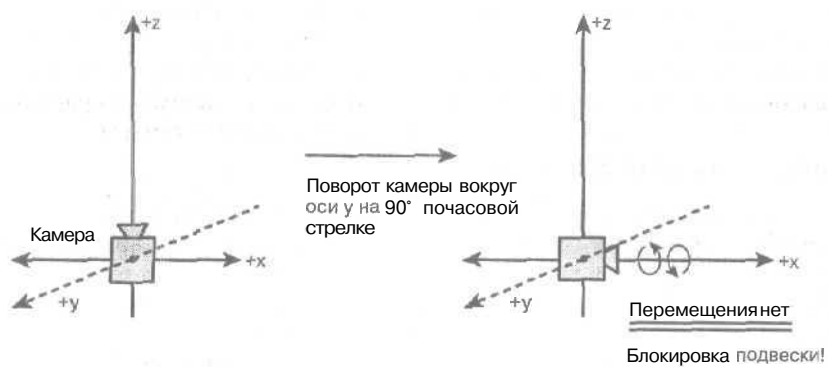


Рис. 6.23. Блокировка подвески

Имеются и другие системы камер, основанные на более естественном представлении, которые лишены этого недостатка (рис. 6.24), но о них мы более детально поговорим в следующей главе.

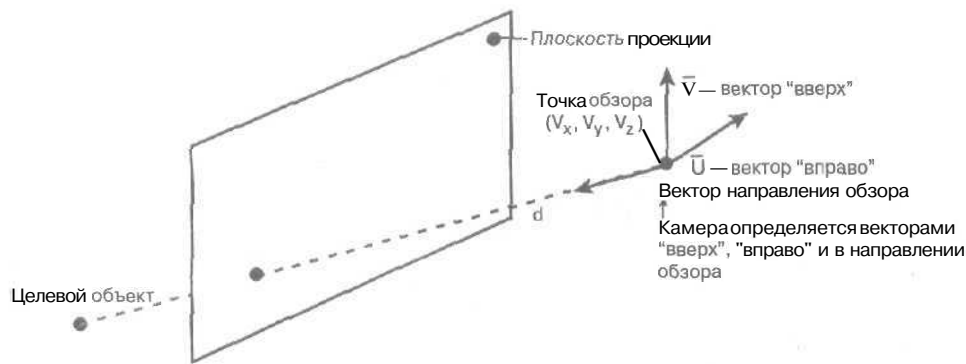


Рис. 6.24. UVN-камера

## Отбраковка скрытых объектов и поверхностей

Все, о чем только что говорилось — вполне корректно и верно, но есть две темы, которые я умышленно не рассматривал, поскольку они не имеют прямого отношения к системам координат; отбраковка скрытых объектов и поверхностей. Оба эти вопроса требуют определенного времени для пояснения, так что **все**, что я намерен сейчас сделать — это дать вам знать, что такой вопрос существует и что он должен рассматриваться в концепции конвейера визуализации.

При выполнении преобразования мировых координат в координаты камеры (или перед ним) можно выполнить ряд тестов для определения того, какие из объектов видимы камере. Эти тесты **обобщенно** классифицируются как отбраковка, или удаление скрытых поверхностей — т.е. нам надо определить, какие объекты не видимы камерой, чтобы не пытаться **визуализировать** их (или, что еще хуже, не поместить их перед камерой). Для этого обычно используются два вида тестов: отбраковка задних поверхностей и проверка описанных сфер.

### Отбраковка задних поверхностей

На рис. 6.25 показан объект и камера, находящиеся в мировой системе координат. Проверка для отбраковки задних поверхностей выполняется до преобразования мировых координат в координаты камеры. Данная проверка позволяет ограничить количество многоугольников, которые должны быть преобразованы в систему координат **камеры**, отбрасывая те многоугольники, которые заведомо невидимы для камеры.

Основная идея проверки заключается в следующем. Все **многоугольники**, составляющие каждый объект, последовательно и согласованно помечаются в некотором порядке (не имеет значения — по или против часовой стрелки). Затем для каждого многоугольника вычисляется нормаль к поверхности  $\mathbf{n}_i$ , которая тестируется при участии вектора обзора так, как показано на рис. 6.25. Если угол между нормалью и вектором обзора меньше  $90^\circ$ , то **многоугольник** видим (очевидно, что данный тест неприменим к **двусторонним** поверхностям). Используя данный тест, мы можем тривиальным образом отбросить (невидимую) половину каждого объекта.

Тест выполняется при помощи скалярного произведения векторов, которое обладает тем свойством, что:

- $\mathbf{i} \cdot \mathbf{v} = 0$  тогда, и только тогда, когда угол между  $\mathbf{i}$  и  $\mathbf{v}$  равен  $90^\circ$ ;
- $\mathbf{i} \cdot \mathbf{v} > 0$  тогда, и только тогда, когда угол между  $\mathbf{i}$  и  $\mathbf{v}$  меньше  $90^\circ$ ;
- $\mathbf{i} \cdot \mathbf{v} < 0$  тогда, и только тогда, когда угол между  $\mathbf{i}$  и  $\mathbf{v}$  больше  $90^\circ$ .

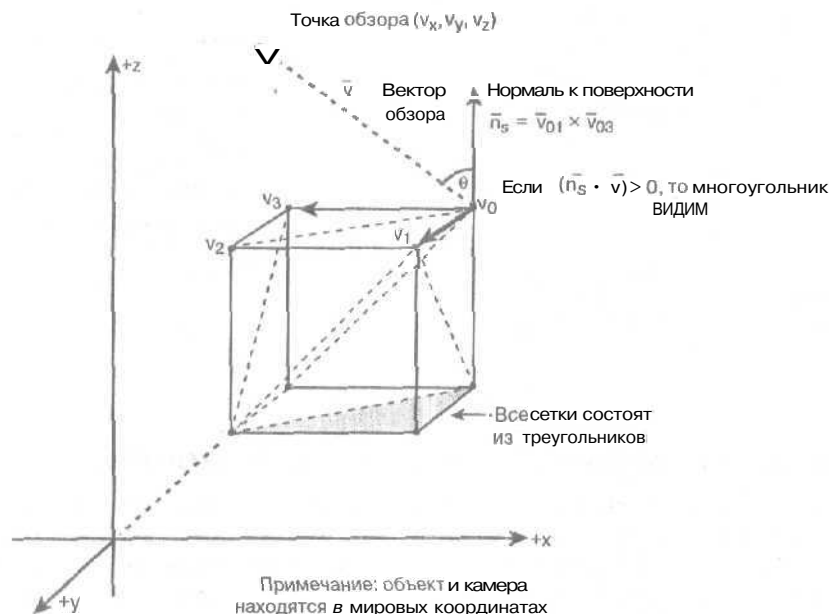


Рис. 6.25. Отбраковка задних поверхностей

Следовательно, в нашей структуре данных, которая определяет объект как состоящую из многоугольников сетку (вскоре мы поближе познакомимся с этим вопросом), должен быть флаг, указывающий, является ли многоугольник односторонним, так что мы можем применить к нему тест для отбраковки задних поверхностей, показанный далее в псевдокоде.

```
if ((n_s · v) > 0)
{
    // Многоугольник видим
}
```

НА ЗАМЕТКУ

Можно решить, что многоугольники, повернутые под углом, в точности равным  $90^\circ$ , видимы, однако такое решение может вызвать определенные проблемы, поскольку многоугольники обычно имеют толщину, равную одному пикселю.

## Проверка ограничивающей сферы

После выполнения отбраковки задних поверхностей и удаления по возможности большего количества многоугольников, мы приступаем к этапу преобразования мировых координат в координаты камеры. Однако еще до выполнения указанной отбраковки можно выполнить еще один тест, причем на более высоком уровне — тест, который позволит отбраковывать не отдельные поверхности, а объекты целиком. Однако этот тест применим только в случае, если объекты разделены по выпуклым областям, или, другими словами, если вы можете выделить отдельный объект и поместить его в охватывающий объект — параллелепипед или сферу. Если же, например, вся игра состоит из единой сетки, то данный тест неприменим. Хотя, конечно, вы можете разбить пространство на части и работать с ними. Цель наших действий очевидна: удалить из обработки большие области игры, которые заведомо не видимы на экране.

Идея проверки состоит в создании ограничивающих каждый объект сфер, как показано на рис. 6.26. После этого выполняется преобразование мировых координат центра

сферы (одной точки) в координаты камеры и проверка, находится ли данная сфера в области обзора камеры. Если **нет** — весь объект в целом можно не обрабатывать. Если же сфера частично содержится в области обзора, результат проверки не окончательный и для данного объекта требуется выполнение дальнейших тестов.

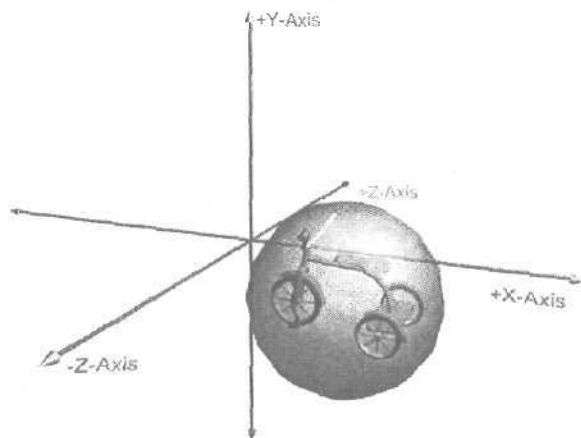


Рис. 6.26. Ограничивающая сфера вокруг объекта

Таким образом, если у нас имеется мир, составленный из объектов, составленных в свою очередь из многоугольников, первым шагом может быть вычисление **охватывающей** сферы вокруг каждого объекта, выполнении преобразования координат и проверка расположения данной сферы в области обзора. С теми объектами, которые остаются не удаленными, выполняется отбраковка задних поверхностей и окончательное **преобразование** мировых координат в координаты камеры.

Хотя я не хочу подробно рассматривать данный алгоритм до тех пор, пока у нас **не** будет работоспособного трехмерного **процессора** для каркасных моделей, мы все же вкратце проанализируем **математический** аспект вопроса. Пусть у нас имеется **некоторый** объект  $O$ , представляющий собой множество вершин. Мы вычисляем окружающую сферу путем определения вершин, наиболее удаленных от центра объекта (что лучше делать не в реальном времени, а на этапе загрузки объекта), что и дает нам максимальный радиус. Далее приведен один из возможных алгоритмов, который в большинстве случаев следует использовать для локальных координат.

```
POINT3D cube_model[NUM_VERTICES]; // Трехмерная модель
```

```
float max_radius = 0; // Изначально равен 0
float curr_radius;
```

```
// Итерации по списку вершин и обновление значения
// максимального радиуса
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
{
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;
    // Найден больший радиус?
    if (curr_radius - sqrt(x*x + y*y + z*z) > max_radius)
```

```

    max_radius = curr_radius;
} // for

Конечно, использования функции sqrt() хотелось бы избежать. И этого легко добиться, если искать вершину с наибольшим квадратом расстояния до центра, а функцию sqrt() использовать только один раз, вынеся ее из цикла.
POINT3D cube_model[NUM_VERTICES]; // Трехмерная модель

```

```

float max_radius = 0; // Изначально равен 0
float curr_radius;

// Итерации по списку вершин и обновление значения
// максимального радиуса
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
{
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;
    // Найден больший радиус?
    if (curr_radius = (x*x + y*y + z*z) > max_radius)
        max_radius = curr_radius;
} // for

// Вычисление реального расстояния
max_radius = sqrt(max_radius);

```

#### СОВЕТ

Использованный метод основан на том, что если  $a > |b|$ , то  $a^2 > b^2$ .

В любом случае это не имеет особого значения, поскольку данный алгоритм применяется только на этапе инициализации. После того как мы найдем `max_radius`, можно выполнить оставшуюся работу по созданию охватывающей сферы. Сперва центр объекта в мировых координатах (точка  $P_0(x_0, y_0, z_0)$ ) преобразуется в координаты камеры:

$P_1 = P_0 \cdot T_{wc}$ . Точка  $P_1$  представляет собой центр ограничивающей сферы в координатах камеры, как показано на рис. 6.27. Далее мы определяем шесть дополнительных к  $P_1$  точек, находящихся на расстоянии `max_radius` от нее в положительных и отрицательных направлениях осей координат. Теперь мы готовы выяснить, находится ли охватывающая сфера внутри области обзора.

На рис. 6.27 приведен вид плоскости  $xz$  сверху. Проверки для плоскости  $yz$  выполняются аналогично, поэтому их мы не рассматриваем. Имеется несколько вариантов отношений охватывающей сферы и области обзора — полное исключение, полное или частичное включение. Если охватывающая сфера  $S$  частично или полностью находится в области обзора, мы должны включить объект в поток данных и в дальнейшем использовать другие тесты. На этом этапе мы можем отбрасывать только те объекты, ограничивающие сферы которых полностью выходят за пределы области обзора.

Итак, задачу можно переформулировать для двумерного случая следующим образом: имеется окружность с центром  $P_1(x_1, z_1)$  и четырьмя точками, определяющими ее границы в направлении осей  $x$  и  $z$ :

```

P1(x1, z1)
P2(x1, z1 + max_radius)

```

$$p_3(x_1, z_1 - \text{max\_radius})$$

$$p_4(x_1 + \text{max\_radius}, z_1)$$

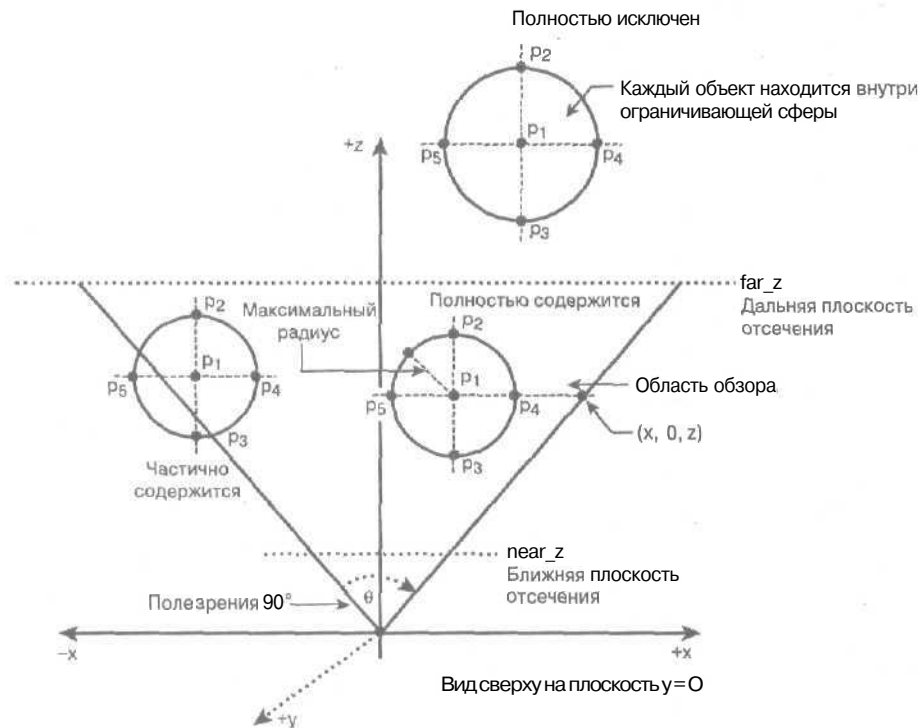
$$p_5(x_1 - \text{max\_radius}, z_1)$$


Рис. 6.27. Проверка ограничивающей сферы

Очевидно, что если хотя бы одна из точек  $p_1 - p_5$  находится в области обзора, то охватывающая окружность (сфера) должна попадать в область обзора. Точный расчет того, принадлежит ли хоть одна точка ограничивающей окружности области видимости, достаточно сложен, и для наших игровых целей можно применить следующее упрощенное правило — если ни одна из точек  $p_1 - p_5$  не попадает в область обзора, то мы считаем, что такой объект должен быть удален.

Начнем с проверки расположения объекта перед ближней или за дальней плоскостями отсечения. Понятно, что данная проверка выглядит следующим образом.

```
if ((p3.z > far_z) || (p2.z < near_z))
{
    // Объект вне пределов области обзора
} // if
```

Это самый простой случай, и всегда поневоле надеешься, что он позволит отбраковать побольше объектов — поскольку из-за большей степени обобщенности боковых граней области обзора остальные проверки несколько сложнее. Для их выполнения можно использовать подобные треугольники или уравнения плоскостей вместе со скалярным произведением векторов.

Однако вспомним, что мы решили ограничиться исключительно простым случаем, а именно углом обзора, равным  $90^\circ$ , и уравнение для боковых плоскостей выглядит как  $|x| = z$ . Таким образом, точка  $p(x,z)$  находится справа от правой плоскости, если  $x > z$ , и слева от нее в противном случае. При рассмотрении левой плоскости следует просто изменить знак  $x$ . Итак, для FOV, равного  $90^\circ$ , проверки выполняются следующим образом.

```
if ((p5.x > p5.z) || (-p4.x > p4.z))
{
    // Объект вне пределов области обзора
} // if
```

НА ЗАМЕТКУ

Все это выглядит очень хорошо, но запомните одну вещь: очень многое зависит от структуры данных и представления объектов. Например, вся ваша игра может состоять из свободных многоугольников, и в ней просто не будет места объектам, которые могут содержаться в области обзора, так что метод ограничивающих сфер оказывается неприменимым. Или, например, все ваши объекты могут быть полупрозрачными, так что все многоугольники имеют две стороны, и бесполезно применять к ним метод отбраковки задних поверхностей.

## Ограничения отбраковки объектов

Перед тем как продолжить изложение материала, я хотел бы обратить ваше внимание на следующий момент. Мы пользуемся правилом — если ограничивающая сфера находится за пределами области обзора, то и объект не может находиться в этой области и не видим для камеры. Однако обратное правило неверно — даже если часть сферы находится в области обзора, сам объект может оставаться невидимым. Почему? Потому что ограничивающая сфера — далеко не всегда лучший способ представления объекта. Например, взгляните на рис. 6.28, где показан очень длинный объект и ограничивающая его сфера. Очевидно, что часть сферы вполне может находиться в области обзора, но при этом сам объект будет оставаться вне этой области. Это не значит, что метод ограничивающей сферы не работает — просто он может иметь не стопроцентную эффективность, только и всего. Иногда более эффективным оказывается применение ограничивающих параллелепипедов.

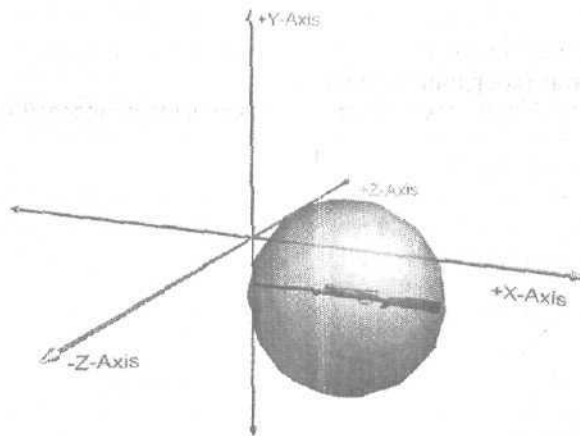


Рис. 6.28. Ограничивающая сфера — далеко не всегда лучший способ представления объекта

На этом наш краткий экскурс в область отбраковки скрытых объектов и поверхностей завершен, и нам остается только уточнить вопрос о конвейере визуализации, показанном на рис. 6.29.

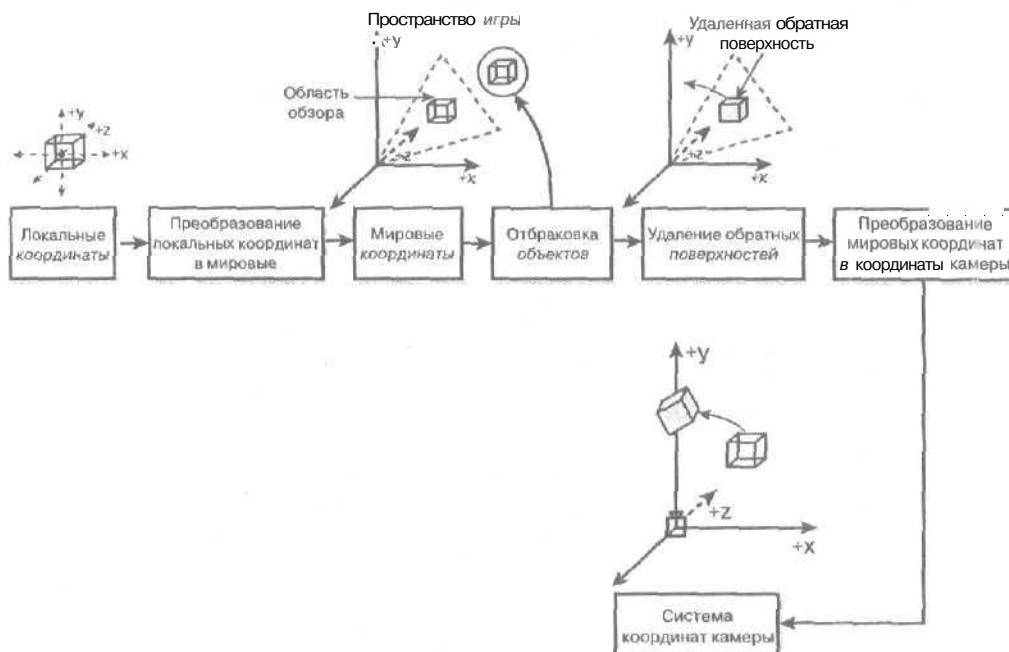


Рис. 6.29. Конвейер визуализации с добавлением отбраковки скрытых объектов и поверхностей

## АксонOMETрические координаты

Итак, мы уже немало добились! Давайте **еще** раз повторим основные этапы конвейера визуализации.

Этап 1. Все трехмерные объекты обычно определяются в своем локальном пространстве и локальной системе координат. Такое представление называется представлением в локальных координатах (координатах модели).

Этап 2. После разработки всех необходимых трехмерных моделей они размещаются в виртуальном трехмерном мире путем преобразования их локальных координат в мировые. Это преобразование представляет собой не что иное, как перенос вершин, который располагает каждый объект в требуемой точке мира. При этом преобразовании надо быть осторожным и не потерять локальные координаты модели, т.е. вам требуется отдельный массив для хранения результатов преобразования локальных координат в мировые.

Этап 3. После того как все объекты размещены в виртуальном трехмерном мире и определены их абсолютные координаты, необходим обзор мира при **помощи** виртуальной камеры. Имеется несколько способов представления камеры, но ее положения и ориентации вполне достаточно **для** наших нужд.

Этап 4. Возможно наличие этапа, когда для минимизации необходимых расчетов производится удаление скрытых объектов и задних поверхностей. Обычно сначала производится удаление целых объектов, а затем — задних поверхностей оставшихся видимых объектов.

Этап 5. Имеется много способов рассчитать вид объекта сквозь линзы виртуальной камеры, но данный процесс существенно **упрощается**, если камера находится в начале

координат и направлена вдоль оси  $z$ . Для этого выполняется преобразование мировых координат объектов в систему координат, связанную с камерой, в которой последняя находится в начале координат и направлена вдоль оси  $z$ . Данное преобразование представляет собой серию преобразований, обратных к преобразованиям, необходимым для размещения и ориентации камеры в пространстве игры.

Этап 6. После того как камера оказывается размещенной в начале координат, а все углы ее ориентации равны 0, мы получаем область обзора, которая представляет собой то, что **видимо** через линзы камеры. Эта область обычно имеет ближнюю и дальнюю плоскости отсечения, наряду с горизонтальными и вертикальными **ограничивающими** плоскостями. Объекты в этой области проецируются и визуализируются на экран. Эта фаза проецирования и представляет собой тему нашего дальнейшего разговора.

## Подготовка к проецированию

Перед тем как приступить к рассмотрению аксонометрических координат, давайте вкратце еще раз поговорим о процессе проекции и области обзора. Предположим, у нас есть трехмерный объект (созданный как набор вершин  $v[]$ ), который мы намерены визуализировать, и мы уже выполнили все необходимые преобразования. Объект находится в области обзора, как показано на рис. 6.18. Вопрос в том, как **спроецировать** этот объект на экран?

Ответ зависит от того, что вы хотите увидеть. Первый шаг состоит в том, чтобы внимательнее присмотреться к системе виртуальной камеры с тем, чтобы немного лучше понять взаимоотношения между камерой, миром игры и экраном.

Вернемся к рис. 6.18, считая, что FOV камеры равно  $90^\circ$  и камера ориентирована вдоль положительного направления оси  $z$ .

## Плоскость проекции

Плоскость **проекции** — это плоскость, **используемая** для формирования виртуального изображения трехмерного мира, которое затем выводится на двумерный экран. Мы уже выбрали систему координат камеры, в которой камера находится в точке  $(0,0,0)$  и направлена вдоль оси  $z$ . Плоскости отсечения и плоскость проекции могут быть выбраны произвольно, однако тот или иной выбор плоскости проекции может как упростить, так и усложнить очередной этап конвейера визуализации.

Например, поскольку мы уже выбрали поле обзора равным  $90^\circ$ , мы можем взглянуть на трехмерную систему сверху без потери общности, как показано на рис. 6.30, и рассмотреть некоторые факты, которые легко обобщаются для трехмерного случая.

Расстояние  $d$  между плоскостью **проекции** и камерой называют расстоянием обзора. Выбор этой величины играет важную роль в аксонометрическом преобразовании. После того как мы разместим объект в области обзора, для его визуализации на плоскости проекции мы **просто** должны математически спроецировать каждую точку объекта на эту плоскость при помощи луча проецирования от точки объекта к точке обзора. Сформированное таким образом на плоскости проекции изображение затем отображается на экране **компьютера**.

Конечно, имеется ряд деталей, которые опущены в этом кратком изложении и к которым мы скоро вернемся. А пока познакомимся поближе с аксонометрическим преобразованием.

## Аксонометрическое преобразование

Это **преобразование** — не что иное, как проецирование вершин объекта на некоторую плоскость **проекции** (рис. 6.31). Его можно сформулировать следующим образом. Дана плоскость проекции, которая находится на расстоянии  $d$  от точки проецирования (точки местонахождения камеры). Требуется математически вычислить точку пересечения прямой, проведенной между точкой проецирования и точкой объекта, и плоскости проецирования.

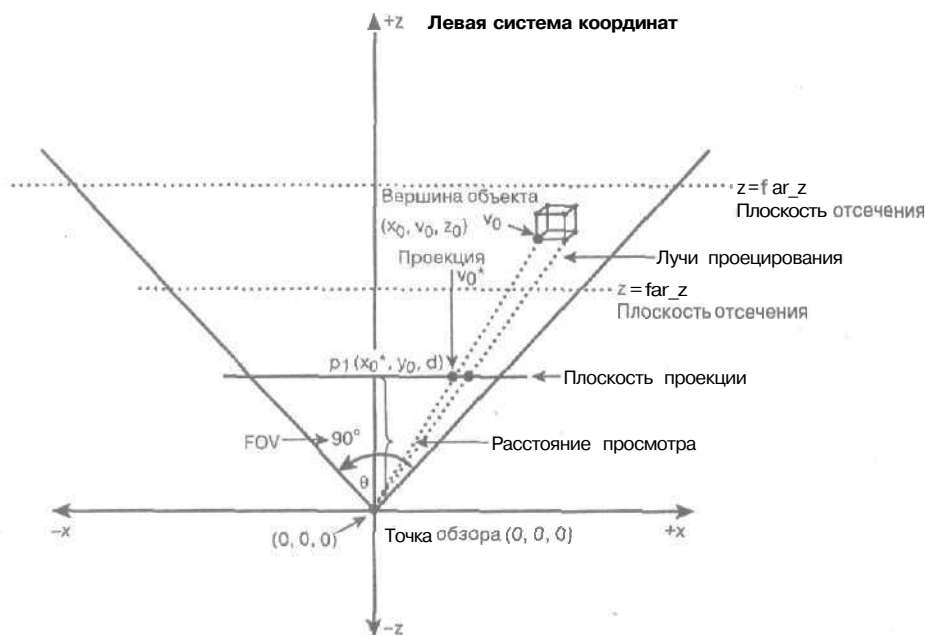


Рис. 6.30. Упрощенная двумерная проекция области обзора на плоскость  $xz$

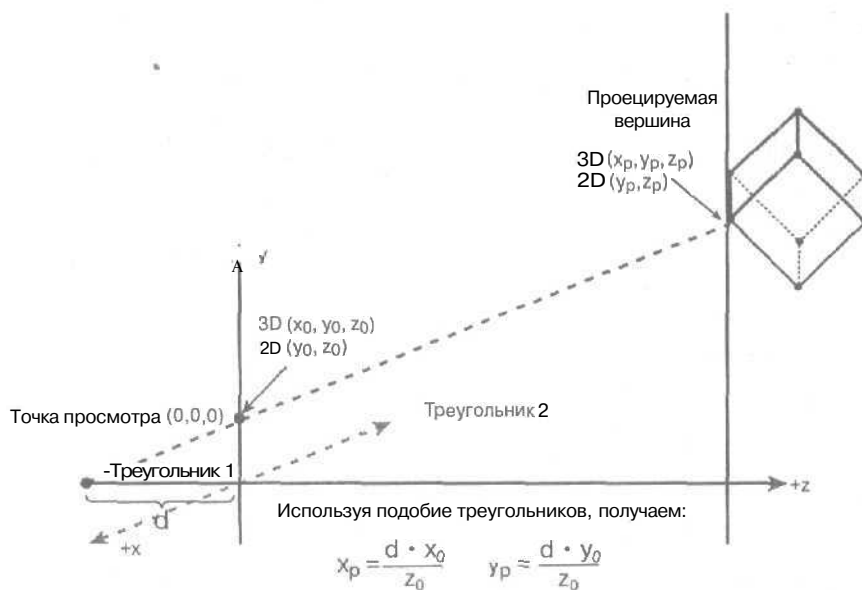


Рис. 6.31. Построение аксонометрического преобразования

Эта задача очень просто решается в двумерном случае и легко обобщается для трехмерного. На рис. 6.31 мы видим нашу трехмерную систему в проекции на плоскость  $yz$ . У нас есть точка  $p(x_0, y_0)$ , которая должна быть спроецирована на плоскость, которая

описывается уравнением  $z = d$ . Рассмотрение тривиальных подобных треугольников позволяет записать пропорцию

$$\frac{d}{z_0} = \frac{y_p}{y_0},$$

что после простейшего преобразования дает искомый ответ

$$y_p = \frac{d \cdot y_0}{z_0}.$$

Анализируя плоскость  $xz$ , можно получить аналогичное соотношение и для координаты  $x$

$$\frac{d}{z_0} = \frac{x_p}{x_0} \text{ и } x_p = \frac{d \cdot x_0}{z_0}.$$

Итак, окончательный вид аксонометрического преобразования для плоскости проекции  $z = d$  и точки обзора  $(0,0,0)$  представлен ниже.

#### Уравнение 6.4. Аксонометрическое преобразование

$$x_{\text{пер}} = d \cdot x / z,$$

$$y_{\text{пер}} = d \cdot y / z.$$

У этих формул есть одна проблема — нулевое значение  $z$ . Увы, проецировать вершины с нулевым значением  $z$  нельзя. Вторая проблема — даже при отрицательном значении  $z$  объект будет спроецирован (хотя и инвертирован). Чтобы всего этого не случилось, и нужна ближняя плоскость отсечения.

##### НА ЗАМЕТКУ

В некоторых программах типа САПР используется параллельная, или ортографическая проекция. При таком преобразовании координата  $z$  просто отбрасывается. Ясно, что эффект перспективы при этом исчезает, однако в программах такой направленности перспектива попросту мешает нормальной работе. Итак, ортографическое преобразование точки  $(x_0, y_0, z_0)$  дает нам точку  $(x_0, y_0)$ , ХОТЯ иногда значение  $z_0$  используется в качестве показателя интенсивности затенения объекта.

Как обычно, рассмотрим аксонометрическое преобразование с точки зрения реализации и в виде матричной операции. Реализующий данное преобразование код очень прост, но для записи преобразования в виде матричной операции мы будем **вынуждены** использовать однородные координаты.

```
POINT3D cube_camera[8]; //Координаты относительно камеры
POINT3D cube_per[8]; //Аксонометрические координаты
```

```
// Выполнение аксонометрического преобразования координат в
// системе отсчета, связанной с камерой, в предположении
// расстояния до плоскости проецирования, равного d
for (int vertex = 0; vertex < 8; vertex++)
{
    float z = cube_model[vertex].z;
    cube_per[vertex].x = d * cube_camera[vertex].x / z;
    cube_per[vertex].y = d * cube_camera[vertex].y / z;
    // Эта координата нам не нужна, так что мы просто
    // копируем ее
    cube_per[vertex].z = cube_camera[vertex].z;
}
> // for
```

Теперь рассмотрим, как реализовать то же при помощи матричного умножения. Соответствующая матрица в однородных координатах имеет следующий вид:

$$T_{\text{пер}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Рассмотрим подробнее преобразование конкретной точки  $p(x_c, y_c, z_c)$ :

$$\begin{aligned} p \cdot T_{\text{пер}} &= [x_c \quad y_c \quad z_c \quad 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix} = \\ &= [x_c \quad y_c \quad z_c \quad (z_c/d)]. \end{aligned}$$

Для того чтобы получить из однородных координат нормальные, нам надо поделить три компонента вектора на значение четвертого, равное  $(z_c/d)$ , так что после преобразования в нормальные трехмерные координаты мы получим точку  $[x_c - d/z_c \quad y_c \cdot d/z_c \quad z_c \cdot d/z_c]$ . Значение координаты  $z$  полученной после проецирования точки нас не интересует, так что мы получаем точку на плоскости с координатами

$$\begin{aligned} x_{\text{пер}} &= d \cdot x_c / z_c, \\ y_{\text{пер}} &= d \cdot y_c / z_c. \end{aligned}$$

Это нам и требовалось. Итак, нам удалось записать аксонометрическое преобразование в виде матричной операции. Впрочем, справедливости ради надо заметить, что это первый случай, когда нам реально требуется преобразование четырехмерных однородных координат в трехмерные. Вы можете спросить — зачем нам нужен этот метод, если возможна и более простая реализация? Дело в том, что, например, наличие матричной формы позволяет в результате выразить все рассмотренные в этой главе преобразования в виде единой матрицы, что существенно снижает количество необходимых вычислений для множества точек.

Еще один вопрос, который заслуживает рассмотрения, — вопрос о расстоянии обзора  $d$ . Каким должно быть данное расстояние? Вопрос интересный, но еще интереснее ответ на него: данное расстояние можно выбрать любым. Изменение  $d$  приводит к изменению масштаба изображений и величины перспективы. И все же, какое значение  $d$  следует выбрать? Я обычно выбираю  $d$  одним из двух способов. Во-первых, если я хочу упростить математические расчеты, я принимаю  $d$  равным 1, что превращает аксонометрическое преобразование в следующее.

#### Уравнение 6.5. Аксонометрическая проекция при $d = 1$

$$\begin{aligned} x_{\text{пер}} &= x/z, \\ y_{\text{пер}} &= y/z. \end{aligned}$$

Матрица для этого частного случая аксонометрического преобразования имеет следующий вид:

$$T_{\text{рег}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Таким образом, мы в конечном счете получаем *нормализованные координаты плоскости проекции*. Другими словами, все проецируемые вершины попадают в диапазон  $[-1, 1]$  как по оси  $x$ , так и по оси  $y$ . При этом дополнительно потребуется отображение координат плоскости проекции в координаты экрана. Но здесь есть одна трудность, которая заключается в том, что исходная область имеет форму квадрата, а значит, *такой* же вид должно иметь и окно обзора на экране, иначе не избежать искажений изображения по одной из координат.

Другой способ поиска  $d$  состоит в вычислении, основанном на окончательных координатах генерируемого экрана игры. Если вы хотите получить, например, экран игры с размерами  $640 \times 480$ , вы можете выбрать  $d$  так, чтобы поле обзора по оси  $x$  составляло  $90^\circ$  и все проекции находились в указанном диапазоне. Поскольку размеры экрана вдоль осей  $x$  и  $y$  отличаются, надо решить, что делать. Использование различных значений  $d$  для оси  $x$  и  $y$  приведет к искажению изображения на экране. Таким образом, если у вас задан определенный размер экрана, **FOV** для оси  $x$  и в результате получается некоторое значение  $d$ , то это значение должно использоваться и для оси  $y$ . В результате мы приходим к тому, что поле обзора по оси  $y$  может отличаться от поля по оси  $x$ .

Таким образом, если вы работаете с неквадратной плоскостью проекции, необходимо помнить о форматном отношении (aspect ratio), которое обычно определяется как отношение ширины к высоте, что в случае экрана  $640 \times 480$  дает 1.33333. Это число приходит на помощь при расчете преобразования аксонометрических координат в экранные для неквадратных плоскостей проекции.

Взгляните на рис. 6.32, на котором показано, каким образом осуществляется вычисление значения  $d$ . Полученное значение  $d$  должно *использоваться* и для вертикальной оси, поскольку у камеры должно быть одно и то же фокусное расстояние в обоих направлениях.

НА ЗАМЕТКУ

В результате в общем случае вертикальное поле обзора отличается от горизонтального, поскольку форматное отношение не равно 1.0.

#### Уравнение 6.6. Определение значения $d$ для экрана размером $W \times H$

$$d = \frac{W}{2} \cdot \operatorname{tg} \left( \frac{\theta_h}{2} \right).$$

В результате для квадратной плоскости проекции формулы остаются такими же, как и раньше.

#### Уравнение 6.7a. Аксонометрическое преобразование для квадратной плоскости проекции

$$x_{\text{рег}} = d \cdot x / z,$$

$$y_{\text{рег}} = d \cdot y / z.$$

Обобщение этих формул должно учитывать форматное отношение, которое мы обозначим как  $ag$ .

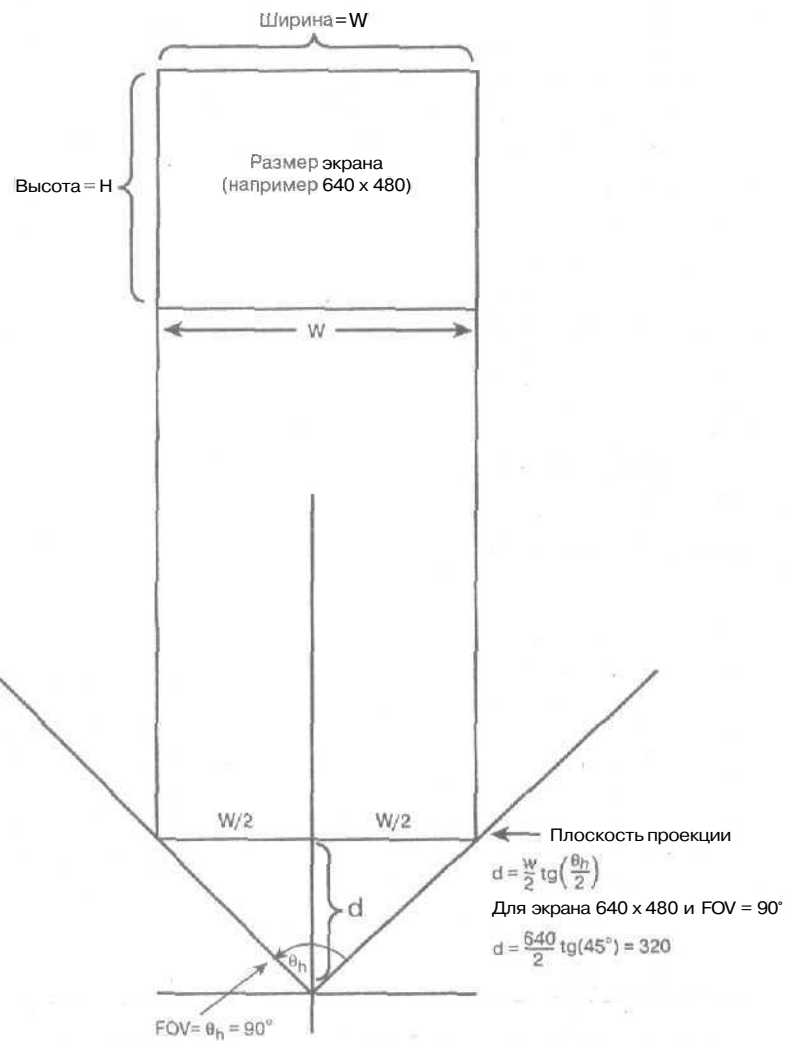


Рис. 6.32. Вычисление  $d$  как функции размера экрана и поля обзора

**Уравнение 6.76. Обобщенное аксонометрическое преобразование для неквадратной плоскости проекции**

$$x_{\text{per}} = d \cdot x/z,$$

$$y_{\text{per}} = d \cdot y \cdot \operatorname{ar}/z.$$

Таким образом, матрица, выполняющая данное преобразование, имеет следующий вид:

$$T_{\text{пер}} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d \cdot \operatorname{ar} & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Убедимся в справедливости приведенной формулы. Пусть дана точка с координатами  $[X \ Y \ Z \ 1]^T$ . Умножим ее на нашу матрицу преобразования и посмотрим, что мы получим в результате.

$$\begin{aligned} P \cdot T_{\text{pers}} &= \\ &= [X \ Y \ Z \ 1]^T \cdot \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d \cdot \ar & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \\ &= [d \cdot X \ d \cdot \ar \cdot Y \ Z \ Z] \end{aligned}$$

Преобразуя четырехмерные однородные координаты в нормальные, получаем  $[d \cdot X/Z \ d \cdot \ar \cdot Y/Z \ 1]$ ,

т.е. идентично уравнению 6.76:

$$\begin{aligned} x_{\text{per}} &= d \cdot X/Z, \\ y_{\text{per}} &= d \cdot Y \cdot \ar/Z. \end{aligned}$$

Главное в том, что мы в состоянии представить аксонометрическое преобразование при помощи матричного умножения. Почему это так важно? Все преобразования, с которыми мы до сих пор имели дело, выражаются при помощи матриц, а значит, результирующее преобразование также можно выразить посредством одной матричной операции, где результирующая матрица представляет собой произведение матриц отдельных преобразований;  $M = M_1 \cdot M_2 \cdot \dots \cdot M_n$ . Значит, вполне возможно объединить преобразование локальных координат в мировые, мировых координат в координаты камеры, аксонометрическое преобразование — словом, все, что мы рассматривали в этой главе, — в одно-единственное преобразование, что позволяет существенно сэкономить время расчетов.

## Аксонометрическая проекция и трехмерное отсечение

Аксонометрическая проекция работает с координатами вершин в системе отсчета, связанной с камерой. Аксонометрическое преобразование координат  $T_{\text{a}}$  (или разные его производные) отображает точки в координатах камеры внутри области обзора на плоскость проекции, подготавливая их вывод на экран (о котором мы еще поговорим). Однако есть еще одно применение аксонометрического преобразования — превращение пирамидальной области обзора в параллелепипед.

На рис. 6.33 показана простая двумерная проекция того, о чем я вам рассказываю. Предположим, что у нас есть две точки  $p_1(x_1, y_1, z_1)$  и  $p_2(x_2, y_2, z_2)$ , определяющие прямую. До сих пор мы не говорили о двумерном или трехмерном отсечении, и теперь настало время и для этого вопроса. Представим, что у нас есть прямая линия, которая проходит через точки  $p_1$  и  $p_2$ , как показано на рис. 6.33, частично располагается в области обзора. Далее предположим, что у нас есть трехмерный процессор для работы с каркасными объектами, так что мы должны вычертить данную линию. Удалить данную линию при помощи отбраковки объектов нельзя, поскольку она не определяет объект и не имеет обратной стороны, поскольку не является многоугольником.

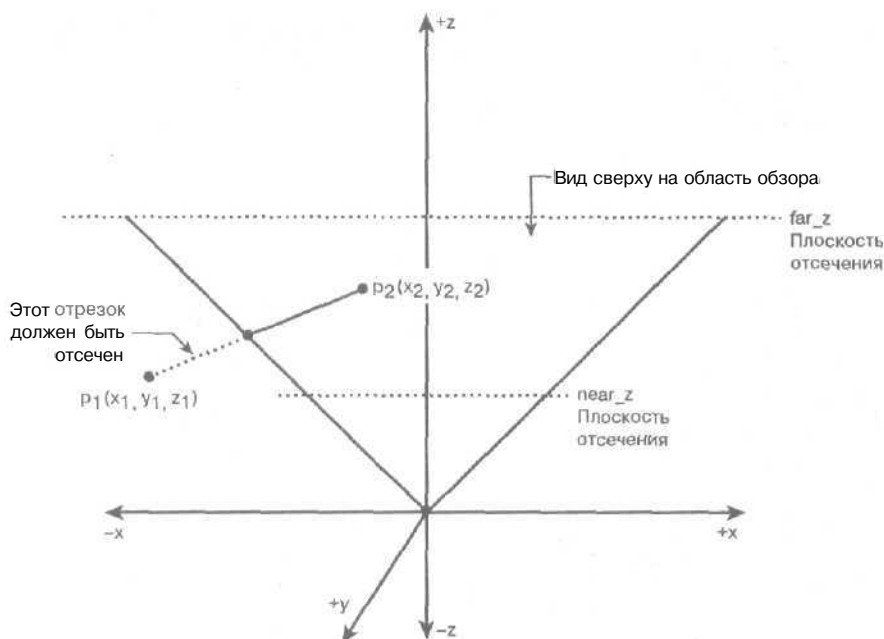


Рис. 6.33. Простейший пример отсечения

Получается, что для того, чтобы начертить данную линию, мы должны прибегнуть к ее отсечению. Этот шаг очень важен, поскольку все объекты, полностью содержащиеся в области обзора, тривиальным образом проецируются и визуализируются. Аналогично, все объекты, которые находятся полностью вне пределов области обзора, тривиально отбрасываются. А вот объекты, которые находятся в области обзора лишь частично, вызывают в связи с необходимостью отсечения наибольшую головную боль.

Есть два способа выполнения отсечения линий. При использовании первого метода мы можем вслепую выполнить аксонометрическое преобразование прямой (считая, что  $z > 0$  для обоих конечных точек), пройти фазу визуализации конвейера и затем выполнить двумерное отсечение в окне. Этот метод называется *отсечением в пространстве изображения* (image space clipping). Другими словами, вы дожидаетесь, пока все объекты, которые должны быть визуализированы, не будут приведены к экранным координатам (хотя эти координаты могут выходить за пределы экрана), и затем отсекаете их в пространстве экрана при помощи окна обзора, как показано на рис. 6.34.

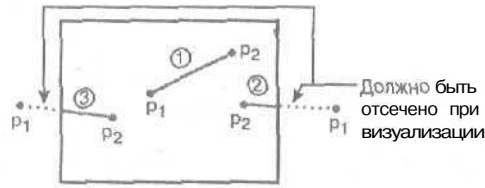
Отсечение в пространстве изображения хорошо работает и легко реализуется, но оно означает, что каждый объект отсекается в двумерном пространстве. Это приводит к сильной загрузке процессора, в особенности при растеризации многоугольников и по-пиксельной проверке. Однако при черчении линий такой метод, вероятно, предпочтительнее отсечения в трехмерном пространстве (о котором я сейчас расскажу).

Второй способ состоит в отсечении в *пространстве объекта*, т.е. в математическом пространстве, в котором существуют объекты. Если помните, мы пытались удалять из конвейера целые объекты и обратные поверхности, чтобы их не надо было даже преобразовывать в координаты камеры. Это была операция в пространстве объектов. Но, естественно, некоторые объекты все же остаются видимыми на экране (иначе что это за игра?). Некоторые из них оказываются не полностью расположенными в обозреваемом объеме и, следовательно, должны быть отсечены. Теперь задача сводится к следующему: перед

тем, как приступить к аксонометрическому преобразованию, все геометрические объекты (прямые и многоугольники), которые лишь частично располагаются в области обзора, должны быть отсечены в трехмерном пространстве областью обзора. Этот процесс называется *трехмерным отсечением*, и он далеко не так прост, как отсечение в пространстве изображения.



Шаг 1. Все прямые (1, 2, 3) имеют координату  $z > 0$  (кроме прямой 4, при проецировании которой возникают проблемы)



Шаг 2. Проецируем все прямые (1, 2, 3) на экранное пространство и выполняем отсечение в процессе визуализации

Рис. 6.34. Отсечение в пространстве изображения

Мы имеем дело с достаточно простой областью обзора, которая ограничена ближней и дальней плоскостями отсечения, боковыми плоскостями по осям  $x$  и  $y$ . Считая, что  $FOV$  равен  $90^\circ$ , получаем квадратную плоскость проекции и уравнения боковых плоскостей  $|x| = z$  и  $|y| = z$ ; в случае прямоугольной плоскости проекции уравнения боковых плоскостей будут следующими:  $x| = z$ ,  $y| = z/\tan(\alpha)$ . Если  $FOV$  не равен  $90^\circ$ , вычисления существенно усложняются.

Суть в том, что если вы первоначально отсечете все частично видимые геометрические объекты в трехмерном пространстве, как показано на рис. 6.33, то все проецируемые объекты будут гарантированно находиться в пределах плоскости проекции. Как я уже говорил, такое *трехмерное отсечение* — задача нетривиальная, и когда немного позже мы займемся ею вплотную, то вы увидите, что отсечению подлежат не только вершины многоугольников, но и текстуры и координаты источников освещения.

Однако отсечение областью, которая является кубом или параллелепипедом, выполняется значительно проще, поскольку по сути представляет собой отсечение прямоугольником в каждой из плоскостей. И теперь я, наконец, могу перейти к главной цели — показать связь между *аксонометрическим преобразованием* и *отсечением*. При выполнении аксонометрического преобразования над всеми потенциально видимыми объектами такое преобразование превращает область обзора в параллелепипед, стороны которого (ранее располагавшиеся под углом  $FOV$  друг к другу) параллельны, и отсечение сводится к отсечению объектов в аксонометрических координатах прямоугольной (в нашем частном случае — кубической) областью. Чтобы было понятнее, рассмотрим пример.

Поскольку все, что будет рассмотрено в одной плоскости, точно так же будет справедливо и для другой, выберем плоскость  $xz$  (рис. 6.35a). Дальняя плоскость отсечения расположена на расстоянии  $z = \text{far\_z}$ , ближняя на расстоянии  $z = \text{near\_z}$ , а плоскость проекции — на расстоянии  $d = 1$ . Угол обзора, само собой разумеется, равен  $90^\circ$ , так что плоскость проекции представляет собой квадрат. Мы хотим выполнить аксонометрическое

преобразование угловых точек пирамиды, которая представляет собой область обзора, и посмотреть, во что превратится эта область после выполнения преобразования.

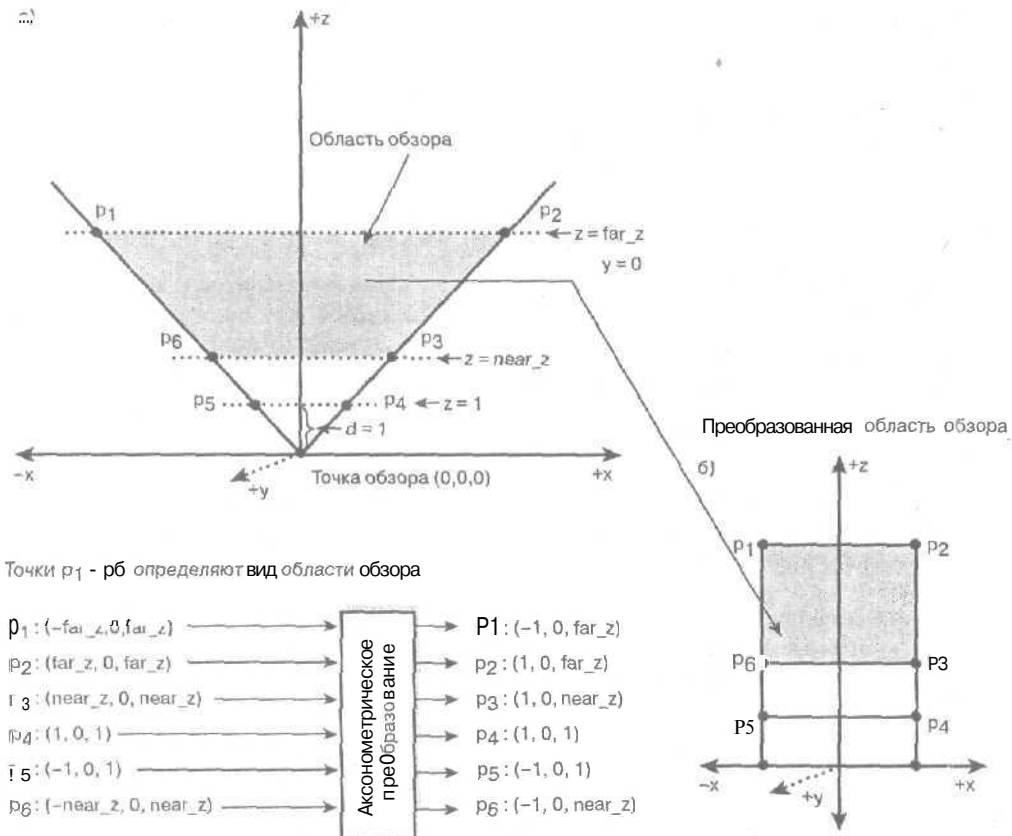


Рис. 6.35. Аксонометрическое преобразование области обзора

Исходные точки  $p_1 - p_6$ , образующие двумерную проекцию области обзора на плоскость  $xz$ , имеют следующие координаты  $p(x, y, z)$ :

$$p_1(-far\_z, 0, far\_z)$$

$$p_2(far\_z, 0, far\_z)$$

$$p_3(near\_z, 0, near\_z)$$

$$p_4(1, 0, 1) \text{ — обратите внимание на то, что координата } x \text{ равна } 1, \text{ т.к. } d=1$$

$$p_5(-1, 0, 1)$$

$$p_6(-near\_z, 0, near\_z)$$

**СОВЕТ**

Вы можете заметить, что координата  $x$  имеет то же абсолютное значение, что и координата  $z$ , что объясняется формулами боковых плоскостей области обзора при  $FOV=90^\circ$ :  $|x| \approx z$  и  $|y| = z$ .

Теперь выполним аксонометрическое преобразование над этими точками при  $d=1$  и получим новые координаты этих точек:

$$\begin{aligned} p'_1(-far\_z/far\_z, 0/far\_z, far\_z) &= (-1, 0, far\_z) \\ p'_2(far\_z/far\_z, 0/far\_z, far\_z) &= (1, 0, far\_z) \\ p'_3(near\_z/near\_z, 0/near\_z, near\_z) &= (1, 0, near\_z) \\ p'_4(1/1, 0/1, 1) &= (1, 0, 1) \\ p'_5(-1/1, 0/1, 1) &= (-1, 0, 1) \\ p'_6(-near\_z/near\_z, 0/near\_z, near\_z) &= (-1, 0, near\_z) \end{aligned}$$

Ясно, что компонента  $y$  везде равна 0, так как мы работаем с плоскостью  $xz$ .

Теперь изобразим на рисунке точки  $p'_1 - p'_6$  (рис. 6.35б). Как видите, они образуют прямоугольник. Итак, мы ухитрились преобразовать жуткую трапецию в приятный прямоугольник, отсечение которым тривиально даже в трехмерном пространстве. Само собой разумеется, мы можем выполнить такой же анализ в плоскости  $yz$  и получить точно такие же результаты. Объединяя их с результатами для плоскости  $xz$ , мы получим область обзора в виде параллелепипеда, как показано на рис. 6.35г. Цель этого рассмотрения — показать, что аксонометрическое преобразование можно использовать также для *нормализации* геометрии области обзора, что делает отсечение почти тривиальной задачей. В последующих главах все это будет опробовано на практике.

В заключение следует добавить, что рассматриваемый нами конвейер *визуализации* получает новые элементы, причем помимо аксонометрического преобразования сюда входит трехмерное отсечение, и конвейер приобретает вид, показанный на рис. 6.36. Заметим, что в зависимости от принятого вами решения об отсечении, есть различные пути передвижения по конвейеру.

Теперь можно приступить к рассмотрению последнего этапа конвейера *визуализации* — экранным координатам.

## Экранные координаты

Перед тем как приступить к описанию последней составной части конвейера *визуализации*, хочу *напомнить*, что здесь мы не рассматриваем вопросы *освещения* и *текстур*. Эти темы будут разбираться позже, а пока достаточно знать, что между использованием мировых и экранных координат мы должны *выполнить все необходимые* действия по обработке *освещения* и *текстурирования* наших многоугольников.

Итак, в настоящий момент у нас есть множество точек, представляющих объекты, которые мы хотим увидеть на *экране*. Эти точки уже прошли аксонометрическое преобразование, но могут быть отсечены из-за размеров экрана, однако в *настоящий* момент вопросами отсечения мы заниматься не будем. Нас интересует другой вопрос, а *именно* — вычисление экранных координат. Это вычисление *основывается* на том, как *именно* проводилось аксонометрическое преобразование, т.е. каким был размер плоскости проекции в виртуальном пространстве? Это очень важно, поскольку мы будем отображать данную плоскость проекции на экран. Рассмотрим несколько возможных случаев.

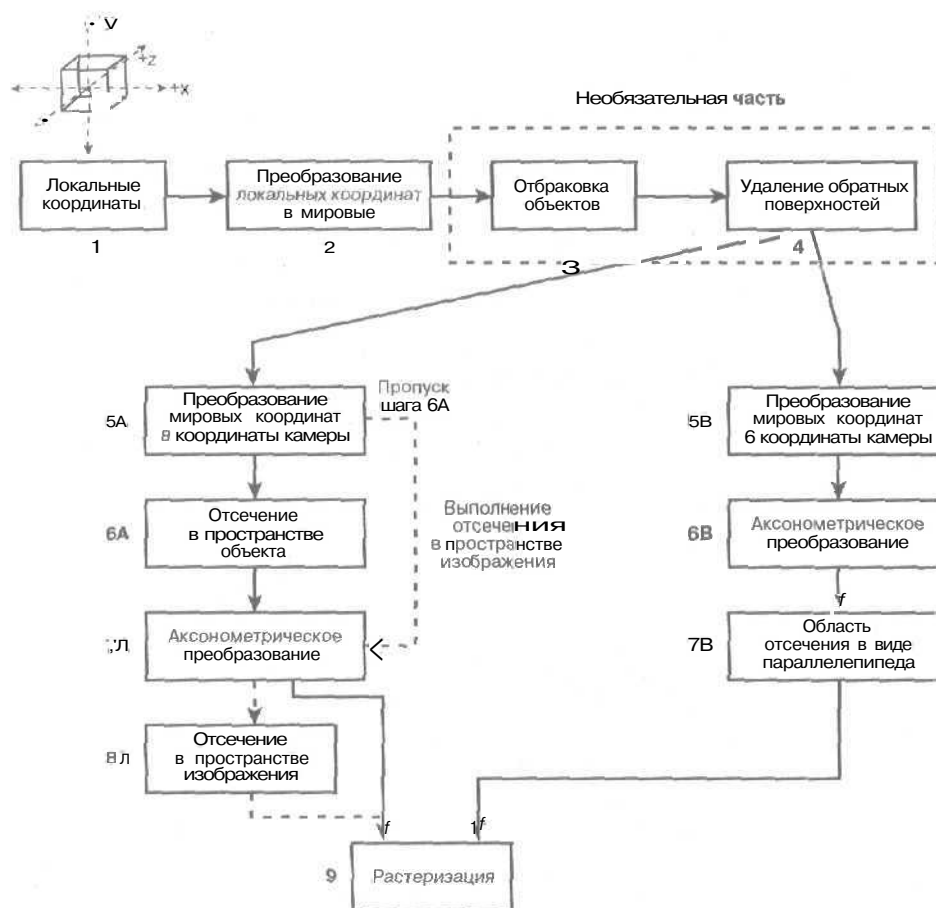
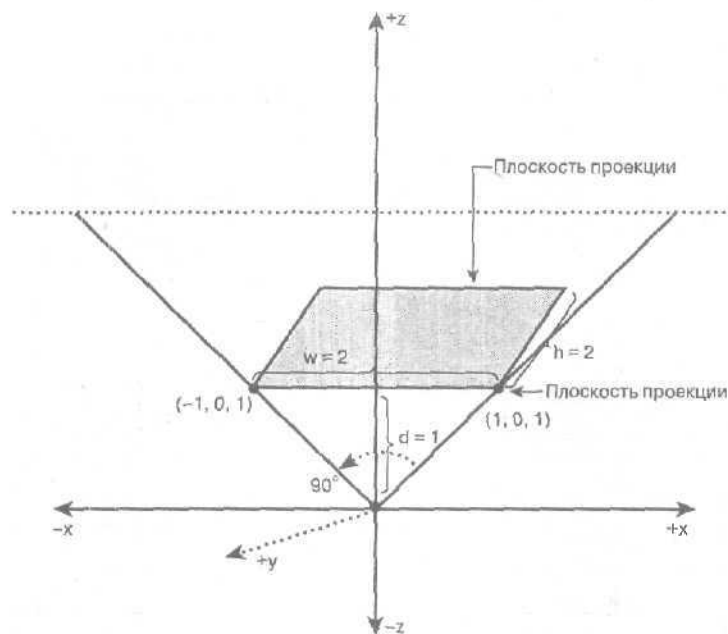


Рис. 6.36. Очередная итерация конвейера визуализации

### Полезрения 90° и единичное расстояние до плоскости проекции

В случае  $FOV=90^\circ$  и расстояния до плоскости проекции  $d=1$  мы получаем квадратную плоскость проекции, и аксонометрическое преобразование трансформирует все объекты таким образом, что они оказываются спроецированы на квадрат размером  $2 \times 2$  ( $x \in [-1, 1]$  и  $y \in [-1, 1]$ ). В более общем случае с неквадратной плоскостью проекции объекты спроецированы на виртуальную плоскость, которая имеет размер  $2 \times (2/ar)$  ( $x \in [-1, 1]$  и  $y \in [-1/ar, 1/ar]$ ), где  $ar$  — форматное отношение. Поскольку мы считаем, что форматное отношение учитывается при аксонометрическом преобразовании, нам не надо беспокоиться о нем при переходе к экранным координатам. Однако забывать о нем нельзя ни в коем случае, иначе ваше изображение будет искажено.

Взгляните на рис. 6.37. Мы хотим отобразить координаты плоскости проекции, которая имеет размер  $2 \times 2$  или  $2 \times (2/ar)$ , в наше окно обзора, т.е. в экранные координаты. Повторим еще раз: мы хотим отобразить плоскость проекции на окно обзора. Вспомните, что плоскость проекции находится в виртуальном пространстве и может иметь как те же размеры (так и иные), что и окно обзора, которое представляет собой окончательное окно растеризации.



При  $FOV = 90^\circ$ ,  $d = 1$  и форматном отношении, равном 1, размер плоскости проекции  $2 \times 2$

При  $FOV = 90^\circ$ ,  $d = 1$  и форматном отношении, не равном 1, размер плоскости проекции  $2 \times 2/\text{ar}$

Рис. 6.37. Проекция при  $FOV = 90^\circ$  и  $d = 1$

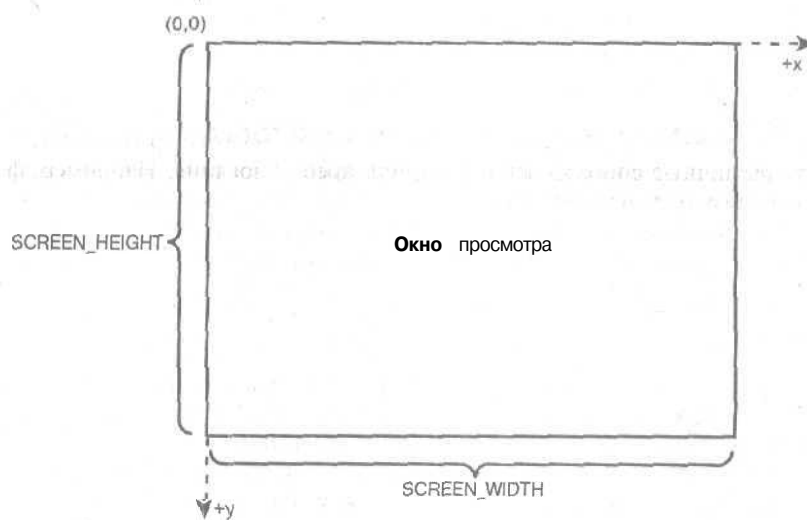


Рис. 6.38. Определение окна обзора

Такое преобразование можно интерпретировать по-разному. Мы будем считать, что визуализация производится в окне обзора с началом координат (0,0) в левом верхнем углу и размерами SCREEN\_WIDTH×SCREEN\_HEIGHT, как показано на рис. 6.38. В нашем случае ширина и высота равны, что соответствует виртуальной системе, которая была разработана нами в главах 2, "Краткий курс Windows и DirectX", и 3, "Виртуальный компьютер для программирования трехмерных игр".

С учетом этого мы просто должны выполнить следующее преобразование плоскости проекции в экранные координаты (которые и есть нашим окном обзора).

$$\begin{aligned}x_{\text{per}} : (-1, 1) &\rightarrow x_{\text{screen}} : (0, \text{SCREEN\_WIDTH} - 1) \\y_{\text{per}} : (-1/\text{ar}, 1/\text{ar}) &\rightarrow y_{\text{screen}} : (\text{SCREEN\_HEIGHT} - 1, 0)\end{aligned}$$

**НА ЗАМЕТКУ**

Обратите внимание на то, что ось  $y$  меняет направление.

Имеется множество способов вывести необходимые математические соотношения, но проще всего, пожалуй, найти центр экрана и выполнить преобразование относительно него. Надо только не забывать об изменении направления оси  $y$ . Впрочем, любой метод приведет нас к одному и тому же результату.

#### Уравнение 6.8. Преобразование в экранные координаты для случая d - 1

$$\begin{aligned}x_{\text{screen}} &= (x_{\text{per}} + 1) \cdot (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5), \\y_{\text{screen}} &= (\text{SCREEN\_HEIGHT} - 1) - (y_{\text{per}} + 1) \cdot (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5).\end{aligned}$$

Проверим приведенные формулы для некоторых граничных условий, а именно — угловых точек плоскости проекции. Выполняя операции, указанные в уравнении 6.8, получим следующие результаты.

$$\begin{aligned}p_1(-1, 1) &= p'_1(0, 0), \\p_2(1, 1) &= p'_2(\text{SCREEN\_WIDTH} - 1, 0), \\p_3(1, -1) &= p'_3(\text{SCREEN\_WIDTH} - 1, \text{SCREEN\_HEIGHT} - 1), \\p_4(-1, -1) &= p'_4(0, \text{SCREEN\_HEIGHT} - 1).\end{aligned}$$

Как видите, мы получили точки, определяющие размер экрана — что и требовалось.

Ясно, что есть различные способы записи формул преобразования. Например, формулу для  $x$  можно записать следующим образом:

$$\begin{aligned}x_{\text{screen}} &= (x_{\text{per}} + 1) \cdot (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5) = \\&= x_{\text{per}} \cdot (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5) + (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5).\end{aligned}$$

Если ввести обозначение  $\alpha = (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5)$ , то мы получим формулу преобразования  $x_{\text{screen}} = \alpha + x_{\text{per}} \cdot \alpha$ .

**Аналогично, для координаты  $y$  получим следующее:**

$$\begin{aligned}y_{\text{screen}} &= (\text{SCREEN\_HEIGHT} - 1) - (y_{\text{per}} + 1) \cdot (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5) = \\&= -y_{\text{per}} \cdot (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5) - \\&\quad - (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5) + (\text{SCREEN\_HEIGHT} - 1) = \\&= -y_{\text{per}} \cdot (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5) + (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5).\end{aligned}$$

что превращается в  $y_{\text{screen}} = \beta - y_{\text{per}} \cdot \beta$ , где  $p = (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5)$ .

Итак, в конечном счете мы получаем

$$x_{\text{screen}} = \alpha + x_{\text{per}} \cdot \alpha,$$

$$y_{\text{screen}} = \beta - y_{\text{per}} \cdot \beta,$$

где

$$a = 0.5 \cdot \text{SCREEN\_WIDTH} - 0.5,$$

$$p = 0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5.$$

Интересно заметить, что точка  $(\alpha, \beta)$  представляет собой центр нашего экрана, что вполне логично, если вспомнить, что на центр экрана отображается точка  $(0, 0)$ . Следует также сказать, что в таком виде формулы легко выражаются при помощи матричной операции. При этом нам не нужны четырехмерные однородные координаты, потому что мы работаем в двумерном пространстве — достаточно использовать матрицу  $3 \times 3$ . Однако все же иногда оказывается удобнее использовать матрицу  $4 \times 4$  (например, для того, чтобы выразить все рассматривавшиеся нами преобразования координат одной матрицей). Я рассмотрю оба варианта, а вы сами выбирайте тот, который больше вам подходит.

Итак, рассмотрим версию преобразования в виде матрицы размером  $4 \times 4$ , полагая, что координаты на плоскости проекции имеют вид  $\begin{bmatrix} x_{\text{per}} & y_{\text{per}} & z & 1 \end{bmatrix}$ , где  $z$  не имеет значения и может быть, например, нулевым, так как служит исключительно для заполнения "лишнего" места. Поскольку данное преобразование выполняется при  $d=1$ , мы назовем соответствующую матрицу  $T_{\text{scr1}}$ :

$$T_{\text{scr1}} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & 0 & 0 & 1 \end{bmatrix}.$$

При умножении этой матрицы на тестовую точку  $p = \begin{bmatrix} x_{\text{per}} & y_{\text{per}} & 0 & 1 \end{bmatrix}$ , мы получим

$$\begin{aligned} p \cdot T_{\text{scr1}} &= \\ &= \begin{bmatrix} x_{\text{per}} & y_{\text{per}} & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & 0 & 0 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} x_{\text{per}} \cdot a + \alpha & -y_{\text{per}} \cdot \beta + \beta & 0 & 1 \end{bmatrix}. \end{aligned}$$

В результате мы получили новые двумерные координаты, связанные с исходными следующим образом:

$$x = \alpha + x_{\text{per}} \cdot a,$$

$$y = \beta - y_{\text{per}} \cdot \beta,$$

т.е. именно то, что и требовалось.

Понятно, что выглядит это на первый взгляд странно: использовать для такого простого и элементарно реализуемого преобразования матрицу  $4 \times 4$ . Но не надо забывать о главной цели использования матриц — выразить при помощи единственной матрицы множество различных преобразований, которые выполняются над каждой точкой.

Перейдем теперь к следующему примеру, а именно — к использованию матрицы 3×3 для преобразования точки в двумерном пространстве с использованием однородных координат. Точка в данном случае имеет вид  $p(x_{\text{per}}, y_{\text{per}}, 1)$ , а матрица преобразования —

$$T_{\text{scr1}} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & -\beta & 0 \\ a & p & 1 \end{bmatrix}.$$

Выполним тестовое умножение;

$$\begin{aligned} p \cdot T_{\text{scr1}} &= \\ &= \begin{bmatrix} x_{\text{per}} & y_{\text{per}} & 1 \end{bmatrix} \cdot \begin{bmatrix} a & 0 & 0 \\ 0 & -\beta & 0 \\ a & p & 1 \end{bmatrix} = \\ &= \begin{bmatrix} x_{\text{per}} \cdot \alpha + \alpha & -y_{\text{per}} \cdot \beta + \beta & 1 \end{bmatrix}. \end{aligned}$$

Итак, в результате выполнения тестового умножения мы получили то же преобразование координат, что и раньше:

$$x = \alpha + x_{\text{per}} \cdot \alpha,$$

$$y = \beta - y_{\text{per}} \cdot \beta.$$

Моя цель достигнута — у вас есть разные способы выполнения преобразований координат. Напоследок я приведу один из вариантов кода, реализующего данное преобразование с нашим кубом, координаты вершин которого после последнего преобразования хранятся в массиве `cube_per[8]`.

```
POINT3D cube_per[8]; // Аксонометрические координаты
POINT2D cube_screen[8]; // Экранные координаты
```

```
// Выполняем преобразование аксонометрических координат в
// экранные, предполагая, что размер экрана равен
// SCREEN_WIDTHxSCREEN_HEIGHT, а расстояние до плоскости
// проекции d = 1.
```

```
// Предвычисление параметров  $\alpha$  и  $\beta$ 
alpha = (0.5*SCREEN_WIDTH-0.5);
beta = (0.5*SCREEN_HEIGHT-0.5);
```

```
// Цикл преобразования
for (int vertex = 0; vertex < 8; vertex++)
{
    cube_screen[vertex].x = alpha + alpha * cube_per[vertex].x;
    cube_screen[vertex].y = beta - beta * cube_per[vertex].y;
} // for
```

Как видите, выполнение данного преобразования без привлечения матриц гораздо проще, чем с ними.

#### НА ЗАМЕТКУ

Не следует забывать, что плоскость проекции находится в виртуальном пространстве, а окно обзора — окно на экране (в большинстве случаев совпадающее с экраном целиком), на которое выполняется отображение плоскости проекции.

## Произвольные поле зрения и расстояние до плоскости проекции

Теперь давайте рассмотрим более общий случай, когда  $d$  вычисляется как

$$d = \text{width} \cdot \operatorname{tg}(\theta_h/2)/2,$$

где  $\text{width}$  — ширина плоскости проекции. Напомню, что в этом случае мы находим  $d$ , основываясь на **главной** оси (обычно ось  $x$ ) поля зрения. Кроме того, мы хотим, чтобы размер плоскости проекции соответствовал размеру экрана, т.е. был равен  $\text{SCREEN\_WIDTH} \times \text{SCREEN\_HEIGHT}$ . Однако нам следует быть осторожными, **поскольку** экран-ные координаты имеют следующий диапазон:

$$x: (0, \text{SCREEN\_WIDTH} - 1),$$

$$y: (0, \text{SCREEN\_HEIGHT} - 1).$$

Таким образом, в предыдущей формуле для расчета  $d$  нам надо **заменить**  $\text{width}$  на  $(\text{SCREEN\_WIDTH} - 1)$ , т.е.  $d = (\text{SCREEN\_WIDTH} - 1) \cdot \operatorname{tg}(\theta_h/2)/2$ . Мы получаем корректное значение  $d$ , если **хотим** объединить аксонометрическое и экранное преобразование в одно. Создается впечатление, что "за бортом" осталась высота экрана  $\text{SCREEN\_HEIGHT}$ , но это не так. Обычно мы сначала проецируем изображение на плоскость проекции с фокальным расстоянием  $d$  для обеих осей — и  $x$ , и  $y$ ; при этом размер плоскости проекции составляет  $x \in [-1, 1]$  и  $y \in [-1/\operatorname{ar}, 1/\operatorname{ar}]$ . После этого, при отображении ее на экран, форматное отношение сокращается. Значит, при отображении **непосредственно** на экран, минуя промежуточный этап плоскости проекции, мы получим

$$x_{\text{per}} = (d \cdot x_c) / z_c,$$

$$y_{\text{per}} = (d \cdot y_c) / z_c.$$

Полученные координаты уже масштабированы к  $(\text{SCREEN\_WIDTH} \times \text{SCREEN\_HEIGHT})$ , так что вы получаете преобразование к экранным координатам непосредственно во время аксонометрического преобразования. Если быть **точным** — то почти **получаете**, поскольку ось  $y$  на экране направлена в другую сторону. Впрочем, это решается очень просто с помощью следующего преобразования;

$$x_{\text{per}} = x_{\text{per}},$$

$$y_{\text{per}} = (\text{SCREEN\_HEIGHT} - 1) - y_{\text{per}}.$$

Очередная проблема немного сложнее, и заключается она в том, что хотя размеры проекции и соответствуют размерам экрана, значения координат по осям  $x$  и  $y$  находятся в следующих диапазонах:

$$x \in [-(\text{SCREEN\_WIDTH} - 1)/2, (\text{SCREEN\_WIDTH} - 1)/2],$$

$$y \in [-(\text{SCREEN\_HEIGHT} - 1)/2, (\text{SCREEN\_HEIGHT} - 1)/2].$$

Решить эту проблему также не представляет особого труда — при помощи простейшего преобразования

$$x_{\text{screen}} = x_{\text{per}} + (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5),$$

$$y_{\text{screen}} = -y_{\text{per}} + (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5),$$

причем данная проблема решается вместе с проблемой обратного направления оси  $y$ .

Если вы проанализируете данные операции, то придете к выводу, что они почти ничем не отличаются от нормализованной проекции экрана без масштабирования. Как

и ранее, обозначая  $\alpha = (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5)$  и  $\beta = (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5)$ , мы получим следующие уравнения преобразования.

#### Уравнение 6.9. Преобразование масштабированных координат плоскости проекции в координаты окна обзора

$$x_{\text{screen}} = x_{\text{per}} + a,$$

$$y_{\text{screen}} = -y_{\text{per}} + b.$$

В матричном виде это преобразование имеет вид

$$T_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}.$$

Предыдущее преобразование отличается от представленного здесь только двумя множителями,  $\alpha$  и  $\beta$ , которые просто масштабируют плоскость проекции, приводя ее к размеру  $2 \cdot \alpha \times |2 \cdot \beta|$ . Итак, главный вывод — мы можем как масштабировать координаты плоскости проекции до экранного преобразования (как часть аксонометрического преобразования), так и масштабировать нормализованные координаты в процессе преобразования аксонометрических координат в экранные.

Моя цель достигнута — вы увидели, что есть два пути выполнения аксонометрического преобразования. Однако обратите внимание на одну маленькую деталь, а именно на преобразование четырехмерных однородных координат в трехмерные в середине процесса. Мы предполагаем, что аксонометрическое и экранное преобразования разбиты на два матричных умножения с промежуточным преобразованием однородных координат каждой вершины в трехмерные координаты. А что, если мы хотим выполнить преобразование из системы координат, связанной с камерой, в экранную систему координат одним действием (как при помощи матрицы, так и вручную)? Давайте сначала посмотрим, как это делается вручную.

Мы должны выполнить аксонометрическое преобразование вместе с масштабированием и переносом, т.е. математически это выглядит следующим образом:

$$x_{\text{screen}} = d \cdot x_{\text{cam}} / z_{\text{cam}} + (0.5 \cdot \text{SCREEN\_WIDTH} - 0.5),$$

$$y_{\text{screen}} = -d \cdot y_{\text{cam}} / z_{\text{cam}} + (0.5 \cdot \text{SCREEN\_HEIGHT} - 0.5),$$

Выглядит тривиально? Именно такая простота и является причиной того, что во многих игровых процессорах не используются матричные преобразования. После того как все вершины будут преобразованы к координатам камеры, для преобразования в экранные координаты можно применить приведенные выше формулы. Однако если вы хотите использовать единственную матричную операцию, то эта задача оказывается нетривиальной в силу того, что у нас имеются четырехмерные однородные координаты и в конце потребуется их преобразование в реальные трехмерные (а в действительности — в двумерные) координаты, которое выполняется путем деления на  $w$ . Помня об этом, рассмотрим матрицу, представляющую собой первую попытку решения задачи:

$$T_{\text{camera}} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & 1 & 1 \\ a & p & 0 & 0 \end{bmatrix}.$$

Все вроде бы выглядит верно. Но давайте применим это преобразование к точке  $p = [x_c \ y_c \ z_c \ 1]$ :

$$p \cdot T_{\text{сдвиг}} = [d \cdot x_c + \alpha \quad -d \cdot y_c + \beta \quad z_c \quad z_c].$$

Хотя результат выглядит на первый взгляд правильным, это только на **первый** взгляд. Дело в том, что при преобразовании однородных координат в обычные трехмерные мы получим соотношения

$$\begin{aligned} x_{\text{экрен}} &= d \cdot x_c / z_c + [\alpha / z_c] \\ y_{\text{экрен}} &= -d \cdot y_c / z_c + [\beta / z_c] \end{aligned} \quad \leftarrow \text{последние члены некорректны!}$$

Эти соотношения не совсем корректны, поскольку при преобразовании однородных координат в нормальные выполняется ненужное нам деление членов  $\alpha$  и  $\beta$  на  $z_c$ . Исправить положение можно, переместив члены  $\alpha$  и  $\beta$  в матрицу преобразования на одну строку вверх:

$$T_{\text{сдвиг}} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ \alpha & \beta & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Теперь при выполнении умножения точки  $p = [x_c \ y_c \ z_c \ 1]$  на данную матрицу будет получен следующий результат;

$$p \cdot T_{\text{сдвиг}} = [d \cdot x_c + z_c \cdot \alpha \quad -d \cdot y_c + z_c \cdot \beta \quad z_c \quad z_c],$$

что после деления на  $z_c$  даст нам требуемое преобразование:

$$\begin{aligned} x_{\text{экрен}} &= d \cdot x_c / z_c + \alpha, \\ y_{\text{экрен}} &= -d \cdot y_c / z_c + \beta. \end{aligned}$$

#### ВНИМАНИЕ

При использовании матриц для аксонометрического и экранного преобразований надо не забывать о том, что в результате получаются однородные координаты, которые необходимо превратить в нормальные трехмерные.

И наконец, вы можете спросить — а куда же делось **форматное** отношение? Да оно просто сократилось при комбинировании аксонометрического и экранного преобразований.

Итак, вопрос в том, какой из описанных методов следует применять? Каждый из них имеет свою область применения. В примерах и демонстрационных программах к **следующей** главе мы будем экспериментировать с каждым из них, но все же, что касается лично меня, то я предпочитаю **использовать**  $FOV=90^\circ$  по оси  $x$  и  $d=1$ .

Как вы видели, у нас есть несколько степеней свободы при выборе угла обзора, фокусного расстояния, ширины и высоты экрана. Однако в большинстве **случаев** мы используем значение  $FOV=90^\circ$  по оси  $x$  и вычисляем значение  $FOV$  для другой оси, исходя из конечных размеров экрана и **того**, что фокусные расстояния вдоль обеих осей одинаковы.

Вы можете решить, что я просто описал весь процесс проецирования, но на самом деле моя цель была несколько иной — я хотел, чтобы вы ясно понимали взаимоотношения между различными системами координат, плоскостью проекции, окном обзора, а также усвоили способы преобразования координат как "вручную", так и при **помощи** матричных операций.

## Окончательный вид конвейера визуализации

Теперь, с учетом последнего рассмотренного материала, окончательный вид конвейера визуализации становится таким, как показано на рис. 6.39. Здесь не рассмотрены вопросы освещения, затенения или текстур, но пока что нас интересуют только геометрические преобразования.

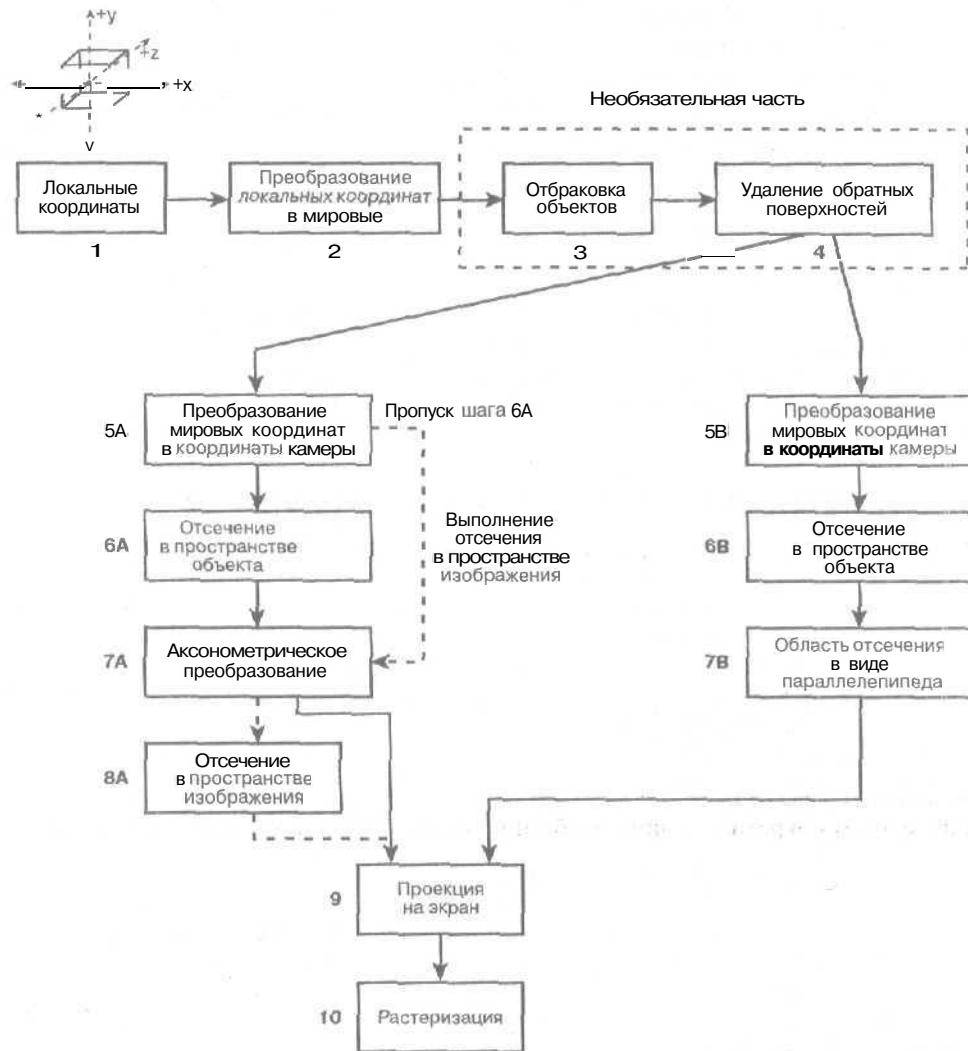


Рис. 6.39. Вид конвейера визуализации после добавления экранного преобразования

## Базовые трехмерные структуры данных

В настоящий момент вы достаточно подготовлены, чтобы перейти к вопросу об общих методах представления трехмерных данных. Вы увидели, что имеется множество преобразований трехмерных данных, как и множество различных представлений потоков трехмер-

ных данных. Следовательно, не существует единственного верного способа выполнить ту или иную работу — есть конкретные способы, работающие в том или ином контексте. Например, способ хранения данных для космической игры, по всей видимости, будет существенно отличаться от способа хранения данных игры в помещении. Кроме того, в зависимости от способа отсечения ваших изображений, из визуализируемых многоугольников могут убираться одни вершины и добавляться другие, что требует определенной предусмотрительности при разработке игры.

Из всего этого следует вывод, что игровой процессор требует серьезного продумывания, и может оказаться, что вы не раз будете переделывать всю сделанную работу, так как обнаружите, что имеется лучший способ реализации той или иной вещи. Подход, использованный мною в данной книге, в достаточной степени общий — мы начнем с некоторых структур данных, которые будем при необходимости совершенствовать. Правильный выбор структур данных будет получаться у вас только тогда, когда вы будете знать о трехмерных технологиях все. Но так как до конца книги еще далеко, сейчас мы только начнем работу со структурами данных с определенными упрощениями.

Я хочу в следующей главе разработать рабочий каркасный трехмерный игровой процессор, который позволит вам перемещать камеру в пространстве. Этот процессор будет способен к минимальному отсечению, которое, вероятно, будет представлять собой некий гибрид отсечения в пространстве объекта и в пространстве изображения. Сейчас же я хочу обсудить некоторые основные структуры данных, которые помогут нам представить трехмерные объекты.

## Представление трехмерных многоугольников

Первое решение, которое вы должны принять при разработке трехмерного процессора, — хотите ли вы поддерживать работу с треугольниками, четырехугольниками или обобщенными  $n$ -угольниками (рис. 6.40). Большая часть трехмерных процессоров (как программных, так и аппаратных) работает только с треугольниками. На то есть много причин, но одна из основных состоит в том, что треугольники проще визуализировать — три точки определяют плоскость, и многие трехмерные алгоритмы лучше всего работают именно с сетками из треугольников.

Конечно, ваши модели, разрабатываемые при помощи соответствующего инструментария, могут иметь обобщенные  $n$ -угольники, но в некоторый момент все модели могут быть триангулированы. Поскольку использование треугольников является обычной практикой, мы с вами будем следовать правилу, что все модели должны быть построены из треугольников или преобразованы к ним перед визуализацией. Пока что мы точно не знаем, как будут получены данные для наших объектов — сгенерированы алгоритмически, введены вручную, загружены из файлов и т.д. Но пока нам не надо об этом беспокоиться — мы просто знаем, что мы работаем с геометрическими объектами, составленными из треугольников.

Даже при таком ограничении имеются проблемы. Взгляните на рис. 6.41, где показаны треугольники, отсекаемые прямоугольной областью. По сути, в процессе отсечения мы можем внести в каждый многоугольник дополнительные вершины. Например, в худшем случае может понадобиться добавление трех новых вершин к треугольнику после его отсечения. Следовательно, нам нужна структура данных, способная справиться с этой задачей.

Вы можете возразить, что можно обойтись без отсечения в пространстве объектов, дождавшись фазы растеризации. Да, это возможно, но тогда вы получите новую проблему, связанную с пересечением ближней плоскости отсечения (и что еще хуже -- с выходом за плоскость  $z=0$ ). Проецирование таких многоугольников может вызвать пробле-

мы, так что нам надо воспользоваться другим способом. Конечно, если все многоугольники малы и их длинные стороны меньше расстояния от ближней плоскости отсечения до плоскости  $z = 0$ , вы можете просто тривиально отбраковать многоугольники из указанного диапазона полностью и не отсекают их, после чего выполнить отсечение в пространстве изображения в процессе растеризации.

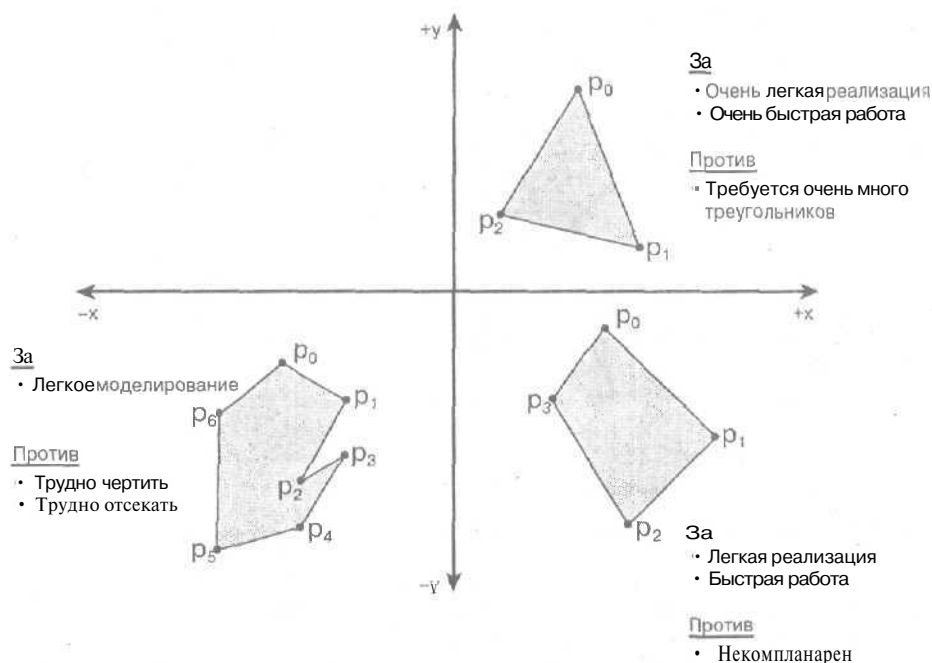


Рис. 6.40. Различные примитивы, используемые трехмерными процессорами



Рис. 6.41. Отсечение треугольника в пространстве объектов создает новые вершины

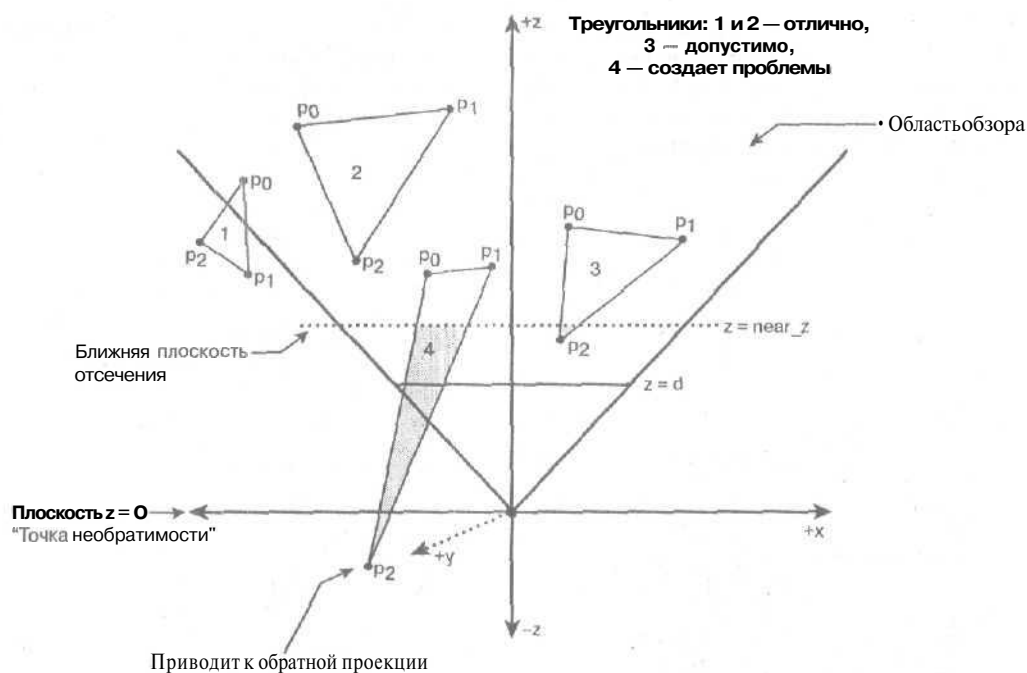


Рис. 6.42. Треугольники, проникающие за ближнюю плоскость отсечения и плоскость  $z=0$

Итак, следует очень тщательно продумывать структуры данных для своей программы. Следует учитывать особенности геометрии пространства и объектов, а также тип используемого отсечения. Эти факторы играют важную роль при выборе представления многоугольников, и они же становятся решающими в случае с многоугольниками, имеющими более чем три вершины.

Можно в принципе воспользоваться правилом, согласно которому все модели объектов строятся с помощью треугольников, а затем, если в процессе работы конвейера визуализации и фазы отсечения появляются многоугольники с более чем тремя вершинами, их можно обрабатывать как частный случай, либо при помощи триангуляции, либо посредством специальной растеризации  $n$ -угольников. Этот подход имеет право на существование, и я тоже использую его. Тем не менее, вы должны рассмотреть все возможные варианты, с учетом всех факторов.

## Определение многоугольников

Наша первая попытка определения многоугольников основана на использовании треугольников в предположении, что никакого отсечения в игровом пространстве не выполняется, так что нет никакого вынужденного добавления вершин. Это означает, что если мы выполняем любое отсечение в игровом пространстве или пространстве камеры, мы либо отбрасываем многоугольник полностью, либо полностью же оставляем его для последующей растеризации (это будет очень легко реализовываться в нашем первом игровом процессоре, который будет работать с каркасными моделями). Данное упрощение позволяет нам продолжить работу без знания алгоритмов трехмерного отсечения. Как бы то ни было, при определении многоугольников мы должны в первую очередь рассмотреть наш трехмерный мир, в частности, определиться с тем, что именно мы будем моделировать.

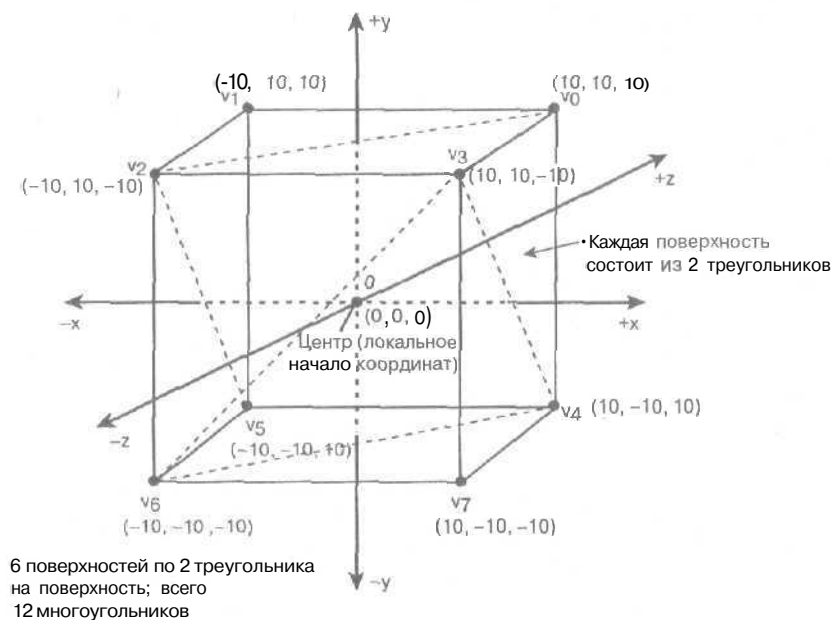


Рис. 6.43. Моделирование куба

Предположим, например, что мы хотим смоделировать трехмерный куб, показанный на рис. 6.43. Куб имеет шесть **поверхностей**, каждая из которых может быть смоделирована как два треугольника (вспомните — четырехугольники нашей моделью не допускаются). Итак, всего 12 многоугольников и 36 вершин. Так? Да, действительно, 12 многоугольников, но вот сколько же на самом деле вершин — 36 или 8? Ответ зависит от того, каким образом вы хотите хранить данную модель. Если вы решите, что каждый многоугольник надо хранить как отдельную **сущность**, то придется определять их как список из трех вершин.

```
struct typedef POLY_EX_TYP_1
{
    POINT3D v[3]; // Список вершин
} POLY_EX_1, *POLY_EX_1_PTR;
```

При этом определение многоугольников будет выглядеть примерно **следующим образом**.

```
POLY_EX_1 face_1[] - { {x0, y0, z0},
    {x1, y1, z1},
    {x2, y2, z2} };
```

Этот метод прост и вполне работоспособен. Его главным недостатком является неоправданный перерасход памяти, поскольку куб имеет всего восемь **вершин**, а при определении куба указанным образом нам потребуется определить 12 многоугольников, каждый из которых имеет по три вершины. Хотелось бы иметь возможность повторного использования **информации** о вершинах, чего можно достичь внесением определенной косвенности в определение **многоугольников** на основе списков вершин.

**Список вершин** представляет собой именно то, что определено его названием — **список** вершин, на которые мы ссылаемся при построении объекта (рис. 6.44). Т.е. мы сначала определяем вершины куба в виде списка или массива, а затем определяем многоугольники при помощи указателей или ссылок на этот массив. Например, список вершин мы можем определить как

```
POINT3D vertex_list[NUM_VERTICES]; // Список вершин
```

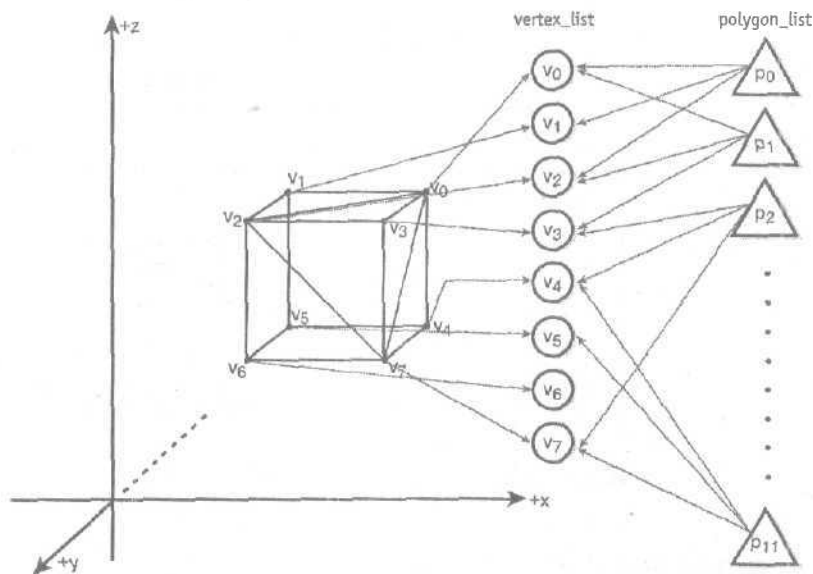


Рис. 6.44. Определение объекта при помощи списка вершин

После чего мы инициализируем каждый элемент массива координатами соответствующей вершины куба.

```
vertex_list[0] = {x0, y0, z0};
vertex_list[1] = {x1, y1, z1};
vertex_list[2] = {x2, y2, z2};
vertex_list[3] = {x3, y3, z3};
vertex_list[4] = {x4, y4, z4};
vertex_list[5] = {x5, y5, z5};
vertex_list[6] = {x6, y6, z6};
vertex_list[7] = {x7, y7, z7};
```

Теперь можно определить **новый тип**, который использует список вершин при определении многоугольника.

```
struct typedef POLY_EX_TYP_2
{
    POINT3D_PTR vlist; // Список вершин
    int vertices[3]; // Индексы в списке вершин
} POLY_EX_2, *POLY_EX_2_PTR;
```

Теперь вы видите, в чем проявляется косвенность. Вместо того, чтобы зря расходовать память на хранение информации об одних и тех же вершинах, имеется единый список вершин для всех многоугольников, так что теперь можно определить отдельный многоугольник, например, следующим образом.

```
POLY_EX_2 face_1 = {vertex_list, 0, 1, 2};
```

Здесь указано, что вершины многоугольника **face\_1** содержатся в массиве **vertex\_list** под номерами 0, 1, 2, т.е. координаты вершин многоугольника **face\_1**, соответственно, равны  $(x_0, y_0, z_0)$ ,  $(x_1, y_1, z_1)$  и  $(x_2, y_2, z_2)$ .

Отметим еще одну важную деталь, а именно — **порядок вершин**. При определении многоугольников или треугольников необходимо придерживаться определенного по-

ряда перечисления вершин, чтобы можно было определить, какая поверхность является *внешней*.

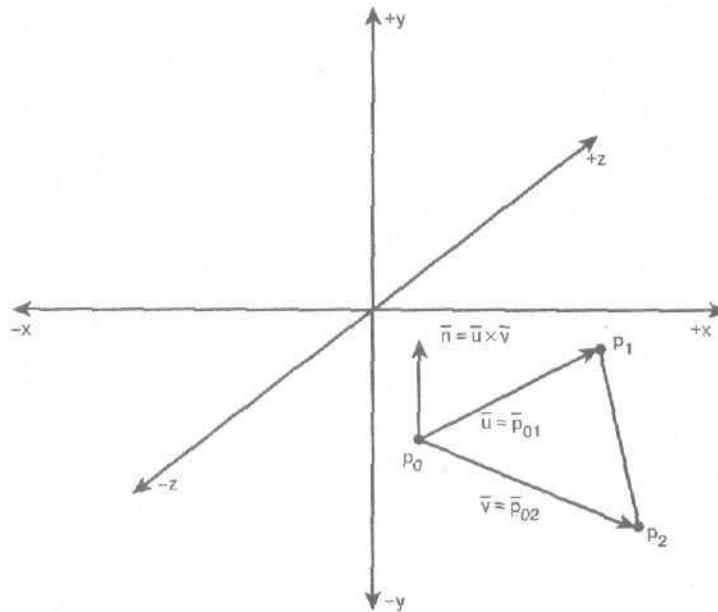


Рис. 6.45. Упорядочивание вершин

Взгляните на рис. 6.45, на котором показан треугольник с вершинами, перечисленными по часовой стрелке. Помните метод отбраковки задних поверхностей? Для его реализации надо обязательно решить, какая сторона многоугольника внешняя, а какая — внутренняя, а затем провести нормаль к многоугольнику (в случае двусторонних **МНОГОУГОЛЬНИКОВ** эта технология неприменима). Для того чтобы сделать это, нам надо использовать схему упорядочения вершин с тем, чтобы мы могли однозначно вычислять внешние нормали к многоугольникам.

При использовании левой системы координат и упорядочения вершин по часовой стрелке, мы можем использовать *правило левой руки* для определения внешнего направления. Другими словами, если мы создадим векторы  $u = p_0 \rightarrow p_1$  и  $v = p_0 \rightarrow p_2$ , и вычислим векторное произведение  $u \times v$ , то получим вектор, нормальный к поверхности многоугольника, как показано на рис. 6.45. Этот вектор нормали  $n$  и есть тот вектор, который используется в методе отбраковки задних поверхностей. В данной книге мы всегда будем использовать упорядочение по часовой стрелке, что позволит нам определить наружную сторону любого многоугольника описанным методом. Однако очевидно, что данный метод справедлив для левой системы координат. При использовании правой системы координат нам надо либо инвертировать знак вектора нормали, либо использовать правило правой руки.

#### СОВЕТ

Заметим, что для произвольного  $n$ -угольника имеется  $n$  различных упорядочений по часовой стрелке, т.е. требуемое упорядочение не является единственным. В принципе лучше использовать упорядочение, когда первыми идут вершины с меньшими номерами. Например, пусть у нас есть упорядоченный список (0,5,12). Тогда удовлетворяющими условию упорядоченности по часовой стрелке будут списки (12,0,5) и (5,12,0). Однако предпочтительнее использовать список (0,5,12), так как зачастую обращение к элементам массива в порядке возрастания индекса имеет более высокую производительность.

Вернемся к списку вершин. Это представление имеет ряд преимуществ. Например, если мы намерены трансформировать объект, нам достаточно изменить информацию о вершинах только в одном списке вершин, и не требуется изменять все вершины в описании каждого многоугольника. Кроме того, если имеется несколько копий вершин — в описании каждого из содержащих ее многоугольников — то при преобразованиях наблюдается постепенное расхождение значений координат этих вершин — хотя бы за счет накапливания ошибок вычислений. В результате оказывается, что в разных многоугольниках одна и та же вершина имеет немного отличающиеся координаты, что приводит к разрушению геометрической целостности объекта.

Теперь пора подумать и о недостатках данного метода. Зачастую при продвижении по конвейеру возникает желание разделить многоугольники и список вершин. На то есть много причин, но для примера представьте, что у вас имеется сетка из 1000 вершин, но при этом только один видимый прямоугольник. Поскольку все преобразования выполняются на уровне списка вершин, вам придется выполнять все необходимые вычисления для всех многоугольников, а не только для видимого. Конечно, есть немало трюков, позволяющих обойти такую ситуацию, но, надеюсь, общий принцип вам понятен? Кроме того, зачастую проще написать растеризатор, который работает с тремя вершинами многоугольника, а не с указателями на элементы списка вершин.

Наконец, на стадии отсечения может оказаться очень трудной работа со списком вершин, а не с многоугольниками. Если вы отсекаете одну вершину, все многоугольники, использующие ее, должны быть отсечены, и это оказывается непростой задачей.

С учетом сказанного выше, мы пришли к использованию гибридного подхода и определения всех преобразований наших объектов на основе списка вершин. Однако иногда мы будем преобразовывать список в отдельные многоугольники, даже если они имеют общие вершины. Мы изменим эту технологию позже, но пока что для простоты будем действовать именно так.

Еще нам следует решить вопрос о том, какие координаты мы будем использовать: трехмерные или однородные четырехмерные? Это трудный вопрос, правильный ответ на который, вероятно, состоит в использовании и тех, и других координат. В конечном счете мы приходим к использованию трехмерных координат, предполагая наличие фиктивного элемента  $w = 1$  и применение специального умножения на матрицы размером  $4 \times 3$ . В настоящий момент, как мне кажется, мы должны использовать четырехмерные координаты и матрицы  $4 \times 4$ , чтобы проиллюстрировать единообразное использование матриц при отсечении, аксонометрическом и прочих преобразованиях. Хотя мы знаем, что нам не надо явно использовать фиктивный элемент и можно обойтись без него, учитывая его наличие в коде, в настоящий момент я не хочу поступать таким образом. Мы примем, что все точки и вершины заданы в четырехмерных однородных координатах с  $w = 1$ , что обеспечивает единообразие всех матричных операций.

## Наш первый многоугольник

Мы не рассматриваем множество деталей описания многоугольников, поскольку пока что мы еще не знаем достаточно для того, чтобы визуализировать их. Однако для следующей главы, в которой мы займемся разработкой каркасного игрового процессора, нам понадобятся, как минимум, следующая информация о каждом многоугольнике.

- **Цвет** — обобщенный цвет многоугольника. Это может быть индекс цвета в 8-битовом режиме или реальное 16- или 24-битовое RGB-значение. Это может быть и значение некоторого пользовательского представления цвета.
- **Состояние** — состояние многоугольника. Здесь хранится информация о динамически изменяющемся состоянии многоугольника в процессе работы с ним

(например, является ли **многоугольник активным**, отсеченным, поврежденным и т.д.).

- Атрибуты — физические свойства многоугольника. Здесь может храниться ряд флагов, описывающих свойства многоугольника — например, его двусторонность, способность к отражению света, прозрачность и т.п.

Напомним, что в соответствии с принятым нами соглашением, все многоугольники никогда не имеют более трех вершин (по крайней мере, пока). С учетом этого вот как выглядит первая версия структуры, описывающей **многоугольник**.

```
typedef struct POLY4DV1_TYP
{
    int state;    // Информация о состоянии
    int attr;     // Физические атрибуты
    int color;    // Цвет

    POINT4D_PTR vlist; // Список вершин
    int vert[3];    // Индексы в списке вершин
} POLY4DV1, *POLY4DV1_PTR;
```

НА ЗАМЕТКУ

При именовании мы придерживаемся следующего соглашения об именовании: [тип][координаты][версия], так что POLY означает тип объекта (многоугольник), 4D — четырехмерные однородные координаты, а v1 — версию.

Ясно, что список вершин `vlist` представляет собой указатель на тип `POINT4D[]`. Пока что мы не будем давать дальнейшее определение полей `state`, `attr` и `color`, так как пока что не собираемся работать с ними. Давайте продолжим и рассмотрим очередную структуру данных, которую я называю *поверхностью* (face).

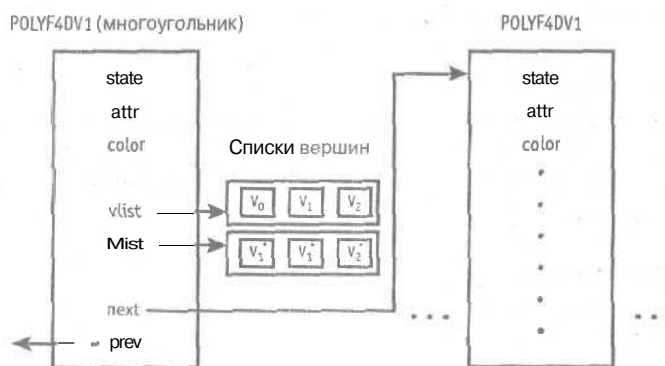
Вспомним, что во время этапа растеризации нам может понадобиться обычный список многоугольников, не связанный с объектами или общим списком вершин. Т.е. нам нужен простой массив или связанный список поверхностей многоугольников, в котором содержится вся необходимая информация и который можно просто передать подпрограмме растеризации. Давайте создадим соответствующую структуру данных **POLYF4DV1** для хранения таких объектов. Она должна быть идентична структуре **POLY4DV1** с тем отличием, что она должна быть самодостаточна и не содержать ссылок на внешний список вершин.

```
typedef struct POLYF4DV1_TYP
{
    int state;    // Информация о состоянии
    int attr;     // Физические атрибуты
    int color;    // Цвет многоугольника

    POINT4D vlist[3]; // Вершины треугольника
    POINT4D tvlist[3]; // Вершины после преобразования
    POLYF4DV1_TYP *next; // Указатель на следующий
                        // многоугольник в списке
    POLYF4DV1_TYP *prev; // Указатель на предыдущий
                        // многоугольник в списке
} POLYF4DV1, *POLYF4DV1_PTR;
```

Обратите внимание на указатели `next` и `prev`, которые позволяют получить связанный список многоугольников наподобие показанного на рис. 6.46, и который мы можем передать растеризатору. Обратите также внимание на две копии списков вершин — `vList[]` и `tvlist[]`. Второй список предназначен для хранения преобразованной версии первого

списка. Позже, возможно, сюда добавится место для хранения мировых координат, координат камеры, аксонометрических и экранных координат.



Лис. 6.46. Структура POLYF4DV1

#### НА ЗАМЕТКУ

Заметим, что мы не имеем права уничтожать исходные локальные координаты объекта при преобразованиях. Мы должны либо скопировать их, либо при преобразовании **сохранять** результаты в другой структуре данных, так как исходные координаты нужны при каждом выполнении преобразований. С другой стороны, при выполнении локальных преобразований наподобие поворота и масштабирования объект обычно таким и остается в дальнейшем, и в этом случае перезапись исходных координат вершин может оказаться оправданной.

Итак, у нас есть список вершин, структуры для хранения многоугольников и поверхностей — вполне достаточно, чтобы начать работать с ними. Теперь давайте повысим уровень иерархии и поговорим об объектах.

## Определение объектов

Объект в трехмерной игре обычно представляет собой набор многоугольников, которые определяют единый объект, как показано на рис. 6.47. Однако в более совершенных трехмерных системах объект может представлять собой набор подобъектов с определенными связями между ними (рис. 6.48). С такими объектами мы будем работать позже, а пока ограничимся простыми объектами, **состоящими** из жесткой сетки многоугольников без подвижных частей, наподобие показанного на рис. 6.47. В этом случае у нас **имеется** объект, который составлен из множества многоугольников на основе списка **вершин**. Такая абстрактная модель позволяет легко разработать структуру данных для хранения объекта. Однако сейчас нам предстоит принять некоторые решения о хранении данных в связи с их использованием в конвейере визуализации.

Вспомним, что у нас есть локальные **координаты**, мировые координаты, координаты камеры, аксонометрические и экранные координаты. Вопрос в том, нужно ли нам хранить каждый набор координат при продвижении по конвейеру? Ответ зависит от множества факторов, но, как минимум, мы должны хранить локальные координаты каждого объекта. Что касается других координат, то ответ на вопрос о них зависит от внутренней **реализации** конвейера. Это означает, что когда мы передаем в конвейер единый объект, то ответственность за хранение результатов преобразований может нести как конвейер, так и сам объект.

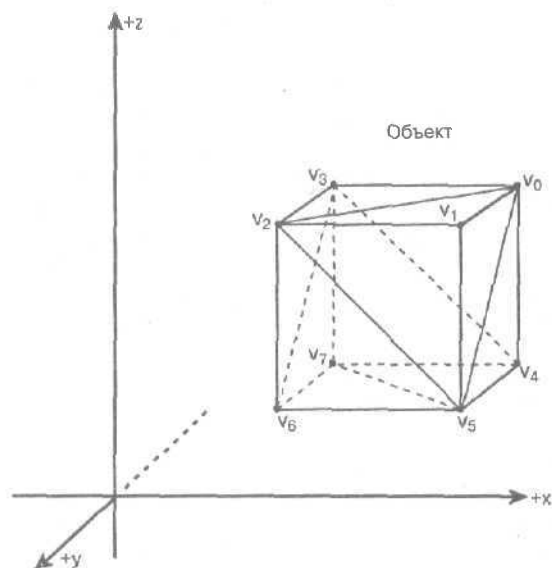


Рис. 6.47, Единый объект, составленный из многоугольников

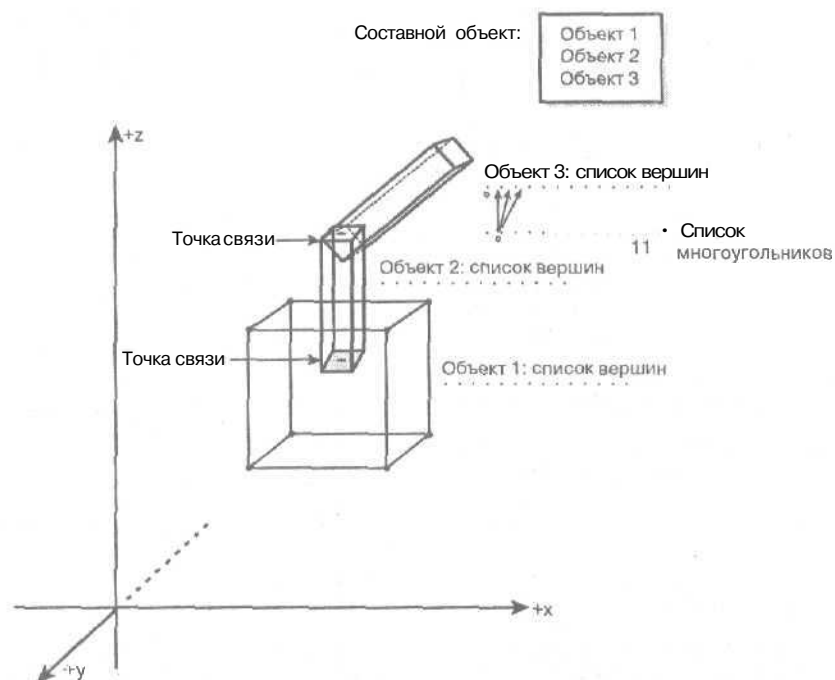


Рис. 6.48. Объект, составленный из нескольких подобъектов

Это сложный вопрос, на который нет однозначного ответа. Я поступал и так, и иначе — т.е. я создавал объекты с выделенной памятью для хранения локальных, **мировых** координат, а также координат камеры. С другой стороны, я также полностью **реализовывал** все это в конвейере.

В настоящее время я хочу обеспечить относительную простоту, так что мы рассматриваем объект как контейнер "всего, чего только можно", включая преобразования конвейера. Однако мы будем хранить только локальные координаты объекта и координаты, полученные после очередного преобразования. После преобразования могут быть получены мировые координаты, координаты камеры, аксонометрические и экранные координаты. Какие именно координаты будут храниться, определяется тем, какая стадия конвейера оказывается выполнена. Таким образом мы минимизируем потребности в памяти. Что еще нам нужно для того, чтобы определить объект? Нам нужно количество вершин, возможно, некоторые атрибуты, состояние объекта, его имя или идентификатор, а также радиус ограничивающей сферы, который понадобится нам позже. Итак, первая попытка определения объекта выглядит следующим образом.

```
// Определение объекта на основе списка вершин
// и списка многоугольников
typedef struct OBJECT4DV1_TYP
{
    int id;                // Числовой идентификатор объекта
    char name[64];         // Имя объекта в виде ASCII-строки
    int state;             // Состояние объекта
    int attr;              // Атрибуты объекта

    float avg_radius;      // Средний радиус объекта для обнаружения
                          // столкновений
    float max_radius;      // Максимальный радиус объекта

    POINT4D world_pos;     // Положение объекта в мире игры

    VECTOR4D dir;          // Углы вращения объекта в локальных
                          // координатах или единичный вектор
                          // направления, задаваемый пользователем

    VECTOR4D ux,uy,uz;     // Локальные оси для отслеживания
                          // ориентации объекта (обновляются
                          // автоматически при вызове функции
                          // поворота)

    int num_vertices;      // Количество вершин данного объекта

    POINT4D vlist_local[64]; // Массив локальных вершин
    POINT4D vlist_trans[64]; // Массив преобразованных вершин

    int num_polys;         // Количество многоугольников в
                          // сетке объекта
    POLY4DV1 plist[128];   // Массив многоугольников
} OBJECT4DV1, *OBJECT4DV1_PTR;
```

Первое, что бросается в глаза, — это невероятное количество напрасно растрачиваемой памяти типом `OBJECT4DV1` из-за использования статических массивов. Но это не является нашим недосмотром; дело в том, что я стараюсь избежать излишнего выделения и освобождения памяти. Очевидно, что более поздние версии будут динамически выделять память для списков вершин и многоугольников. Однако сейчас вершин (64) и многоугольников (128) должно быть более чем достаточно для моделирования интересных нас объектов. Заметим также, что в структуре имеются поля для среднего и максимального радиуса объекта, что играет свою роль при отбраковке объектов.

Само собой, в структуре представлены положение объекта в игровом пространстве `world_pos` и ориентация объекта в локальном пространстве `dir`. Как вы видели, при изменении ориентации объекта в локальной системе координат мы будем изменять локальные координаты и тем самым терять их исходные значения, так что нам необходимо определить метод для отслеживания этих изменений. Конечно, мы можем использовать соглашение, по которому всякий раз при обработке объекта конвейером визуализации мы выполняем локальные повороты "на лету", оставляя исходные значения нетронутыми — но это может привести к существенным затратам процессорного времени при обработке, например, пустыни с 1000 скалами, которые никогда не поворачиваются. Во многих случаях трехмерные процессоры поворачивают объекты в их локальном пространстве и перезаписывают локальные координаты, но такие повороты должны отслеживаться — для чего и предназначено поле `dir`.

Кроме того, мы можем хранить здесь углы или даже единичный вектор, изначально направленный вдоль оси  $z$  (`dir = <0,0,1>`) и поворачивающийся каждый раз вместе с объектом, так что мы можем отследить ориентацию объекта относительно ориентации при начальной загрузке. Кроме того, ориентацию объекта отслеживают и векторы базиса  $ix$ ,  $iy$  и  $iz$ . При каждом преобразовании объекта эти векторы отслеживают его ориентацию. Предположим, например, что мы хотим определить куб (с которым мы уже имели дело с десятком раз) при помощи следующего списка вершин.

```
p0( 10, 10, 10)
p3(-10, 10, 10)
p2(-10, 10,-10)
p1( 10, 10,-10)
p4( 10,-10, 10)
p5(-10,-10, 10)
p6(-10,-10,-10)
p7( 10,-10,-10)
```

Ниже показан код, который выполняет поставленную задачу.

```
// Определение куба вручную (очень утомительное)
OBJECT3DV1 cube_1;

cube_1.id = 0; // В настоящий момент не используется
cube_1.state = 0; // В настоящий момент не используется
cube_1.attr = 0; // В настоящий момент не используется

cube_1.avg_radius = 17.3; // Средний радиус объекта
cube_1.max_radius = 17.3; // Максимальный радиус объекта
// sqrt(10*10+10*10+10*10)

cube_1.world_pos.x = 0; // Положение объекта в мире игры
cube_1.world_pos.y = 0;
cube_1.world_pos.z = 0;
cube_1.local_ang.x = 0; // Текущая ориентация объекта
cube_1.local_ang.y = 0; // (используется по умолчанию)
cube_1.local_ang.z = 1; // для всех объектов

cube_1.num_vertices = 8; // Количество вершин

// Создание списка вершин для упрощения инициализации
```

```

POINT3D temp_verts[8] = { { 10, 10, 10}, // p0
    { 10, 10,-10}, // p1
    {-10, 10,-10}, // p2
    {-10, 10, 10}, // p3
    { 10,-10, 10}, // p4
    {-10,-10, 10}, // p5
    {-10,-10,-10}, // p6
    { 10,-10,-10}}; // p7

// Определение локальных вершин
for (int vertex=0; vertex < 8; vertex++)
{
    cube_1.vlist_local[vertex].x = temp_verts[vertex].x;
    cube_1.vlist_local[vertex].y = temp_verts[vertex].y;
    cube_1.vlist_local[vertex].z = temp_verts[vertex].z;
    cube_1.vlist_local[vertex].w = 1;
} // for index

cube_1.num_polys = 12; // Количество многоугольников

// Определение многоугольников. В данном примере я оставляю
// поля state, attr и color равными 0, а указатель указывает
// на локальные координаты. На самом деле это не имеет
// значения, на какой именно список он указывает, поскольку
// мы знаем, что все многоугольники, определяемые в данном
// объекте, используют его список вершин и не могут
// использовать никакие внешние данные

// Создаем список триад, определяющих каждый многоугольник,
// что упростит нашу задачу инициализации. Обратите
// внимание, что везде использовано упорядочение по часовой
// стрелке
int temp_poly_indices[12*3] = {
    0,1,2, 0,2,3, // Многоугольники 0 и 1
    0,7,1, 0,4,7, // Многоугольники 2 и 3
    1,7,6, 1,6,2, // Многоугольники 4 и 5
    2,6,5, 2,3,5, // Многоугольники 6 и 7
    0,5,4, 0,3,5, // Многоугольники 8 и 9
    5,6,7, 4,5,7}; // Многоугольники 10 и 11

// Инициализируем все треугольники
for (int tri=0; tri < 12; tri++)
{
    // Пока что эти поля не используются
    cube_1.plist[tri].state = 0;
    cube_1.plist[tri].attr = 0;
    cube_1.plist[tri].color = 0;

    // Указывает на список вершин в локальных координатах
    cube_1.plist[tri].vlist = cube_1.vlist_local;

    // Присвоение индексов вершин
    cube_1.plist[tri].vert[0] = temp_poly_indices[tri*3+0];

```

```
cube_1.plist[tri].vert[1] = temp_poly_indices[tri*3+1];
cube_1.plist[tri].vert[2] = temp_poly_indices[tri*3+2];
} // for
```

Однако, если поступать описанным способом, нам понадобятся буквально мегабайты кода! Это одна из наибольших проблем трехмерной графики: огромное количество **данных**. Вы никогда не представляете в игре растровое изображение как набор отдельных байтов? Вот так же вы не должны представлять каждый объект как данные в реальной трехмерной игре. Сетки и игровое пространство всегда загружаются из файлов, поскольку они могут состоять из сотен, тысяч или даже миллионов вершин. Мы используем ввод вручную из некоторых примитивных текстовых форматов для представления трехмерных объектов только на данном этапе, пока не научимся работать с более сложными форматами.

Итак, в **настоящий** момент у нас есть **куб**, и мы можем провести его по конвейеру визуализации — обо всем этом вы уже знаете. Осталось последнее действие — непосредственная визуализация **объекта**, затененными многоугольниками или каркасная, но об этом мы поговорим в следующей главе.

## Представление миров

Итак, в **настоящий** момент наш трехмерный игровой мир представляет собой композицию из трехмерных объектов, в котором каждый объект представляет собой набор многоугольников, основанный на списке вершин (а может, и нет). Все это хорошо, но что если мы хотим создать игру типа *Quake* или танковый имитатор с огромным количеством ландшафтов? Конечно, объекты игры легко отображаются на наши объекты, но внутреннее окружение *Quake* или огромные карты местности могут не **помещаться** в наше небольшое определение объекта.

Внутренние и внешние миры (или, говоря более обобщенно, окружения, или среды) обычно представляют собой частные случаи объектов, и для их корректного представления могут быть разработаны более совершенные структуры **данных**. Например, **типичный** объект игры состоит из порядка 20-500 многоугольников. Такой объект вполне может быть обработан при помощи разработанной нами структуры данных. С другой стороны, представим игровой мир, **состоящий** из десятков тысяч, даже из десятков миллионов многоугольников. Мы не можем поддерживать такой огромный список многоугольников и передавать его конвейеру визуализации. Нам нужен какой-то разумный метод загрузки данных, поскольку в каждый момент игры видна только малая доля многоугольников. О представлении игровых миров уже немного говорилось раньше, и одно из решений состоит в **преобразовании** игрового мира в матрицу секторов. Некоторые из наших алгоритмов (отбраковки объектов) оказываются неприменимы, поскольку не имеет смысла работать с местностями или окружением, как с объектами. С другой стороны, концепции объектов заменяются концепциями помещений, **площадей** и **порталов**, и мы можем разработать и использовать похожие алгоритмы для работы с ними.

Кроме того, местности и **окружения**, как правило, изменяются в процессе игры не очень сильно, так что мы можем выполнить большое количество предварительных вычислений и оптимизаций и привлечь бинарное разделение пространства, деревья и т.п. методы для упрощения и минимизации количества обрабатываемых данных. Например, среда может быть полностью определена в мировых координатах, поскольку нам не надо ее **перемещать**, и это позволяет нам сэкономить на преобразовании локальных координат в мировые.

Таким образом, в большинстве случаев после того, как мы решим, что некоторая часть окружения видима, мы просто передаем ее матрицам преобразования в координаты камеры и аксонометрические координаты.

По ходу книги мы рассмотрим все эти **концепции**, но я бы хотел, чтобы вы поняли, что мы уже можем разработать простую трехмерную каркасную (или с использованием закрашенных многоугольников) игру. Для более сложных игр мы должны подумать о более мощных структурах данных и алгоритмах для уменьшения количества обрабатываемых данных при формировании каждого кадра.

С учетом этого нет причин даже пытаться использовать имеющиеся в нашем распоряжении в данный момент трехмерные структуры для ифовых миров в целом, да и знаний о трехмерных технологиях у нас для этого пока что не хватает. Однако можно представить, что для **игры** типа *Quake* можно разработать такой процессор, что каждое помещение будет представлять собой отдельный узел дерева. Для каждого помещения имеются связи, указывающие, какие помещения могут быть видимы из данного. Таким образом в любой момент времени можно обратиться к нашей структуре данных и на основании нашего **положения** в игровом мире отбросить большинство помещений как невидимые (рис. 6.49).

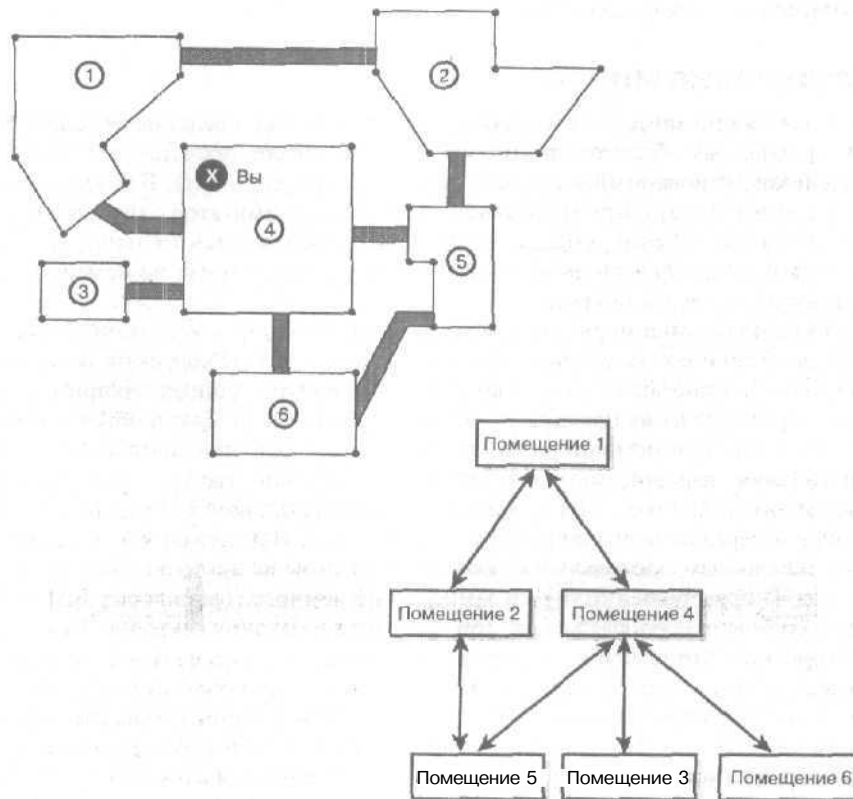


Рис. 6.49. Возможная структура данных уровня игры

Конечно, такой способ оказывается неработоспособным, если мы можем выглянуть наружу или имеется много окон и т.д. Но если вы пишете игру с помещениями, соединенными длинными коридорами, у вас могут быть миллионы помещений и ифовой процессор должен **иметь** возможность отбросить все лишнее. Однако об этом мы поговорим немного позже...

## Инструментарий трехмерного моделирования

Создание трехмерной игры — это в первую очередь большой объем работы, в основном из-за ее трехмерности и огромного количества данных. Именно генерация всех необходимых данных является самой трудоемкой частью разработки игры. Имеется множество решений данной проблемы. В случае простых трехмерных игр можно разработать собственный инструментарий для трехмерного моделирования и создания игровых миров. Например, на рис. 6.50 показана копия экрана старенького редактора уровней *Doom*. Как видите, этот редактор работает как с растровыми изображениями, так и с векторной графикой, и просто показывает план уровня сверху и позволяет его редактировать на уровне отдельных линий и вершин. Он читает различные файлы данных и позволяет вам размещать объекты, изменять геометрию уровня, помещать телепортаторы, персонажей игры, источники света и т.п. для создания собственного игрового мира. *DoomEdit* и другие “процессоры построения миров” использовались для генерации трехмерных миров игр и представляли собой решения типа “все в одном”. Ясно, что другие данные игр — сценарии, растровые изображения, звуки — создавались при помощи другого инструментария.

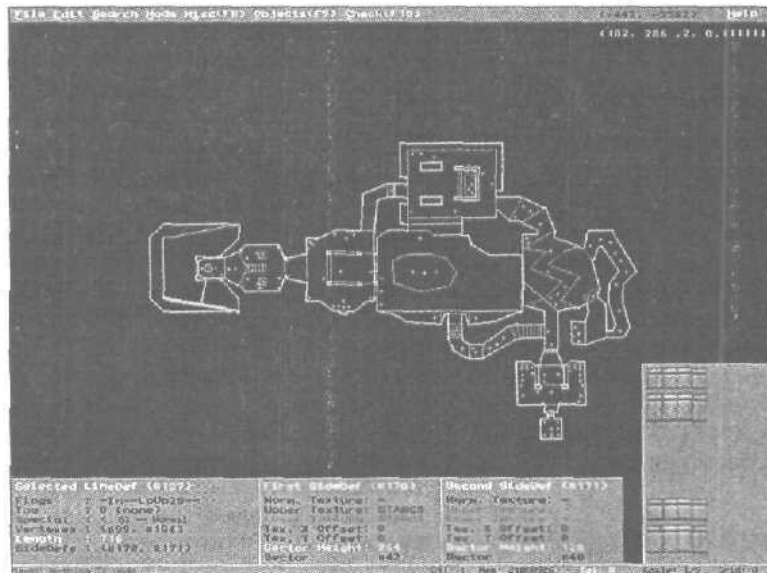


Рис. 6.50. *DoomEdit* в действии

В настоящее время задачи, стоящие перед программистом и дизайнером, существенно усложняются, поскольку все миры стали исключительно трехмерными, с произвольной геометрией. С учетом этого есть два варианта действий: можно писать собственный инструментарий для трехмерного моделирования, а можно воспользоваться готовыми программами наподобие *Caligari trueSpace*, *Maya*, *LightWave*, *3D Studio Max*, *Blender* или *Mogau*, полностью разработать свои объекты и миры с их помощью, а затем использовать их в своей программе при помощи дополнительного программного кода. Вероятно, это наилучший выход из положения, поскольку вы можете привлечь к своей работе кого-нибудь еще. С другой стороны, такие программы трехмерного моделирования не позволяют вам размещать монстров, добавлять переключатели для дверей и т.п., но многие из

упомянутых программ моделирования обеспечивают возможность добавления обработчиков и дополнительные API, что позволит вам изменять и дополнять базовую функциональность программы, адаптируя ее согласно своим целям.

Изначально я планировал использовать в этой книге формат *Quake* для трехмерных миров и воспользоваться инструментарием наподобие *WorldCraft* для их моделирования, однако разработчики *WorldCraft* удалились от дел, передав программу фирме *Valve, Inc.* (теперь Программа известна под названием *Hammer Editor*), так что формат *Quake* представляется частною собственностью и, по здравому размышлению, я предпочел с ним не связываться. Мы будем использовать один или несколько из стандартных форматов файлов типа *Caligari .COB*, *3D Studio Max .3DS* или *.MAX*, *.DXF*, *.NFF* и дескриптор, чтобы вы могли использовать распространенный инструментарий трехмерного моделирования при разработке собственных игр. Кроме того, это книга о трехмерной графике и программировании игр, а не о разработке уровней, так что мы не будем концентрировать внимание на этом вопросе. Главное, что следует помнить, — вы сможете использовать любой из инструментов, если он создает файлы в формате, рассмотренном в данной книге.

Ряд демонстрационных программ трехмерного моделирования вы сможете найти на прилагаемом компакт-диске, где также имеются некоторые условно-бесплатные версии некоторых программ. Кроме того, там же имеется ряд программ для преобразования данных из одного формата в другой.

## Анимационные данные

На некоторой стадии разработки игры вы можете захотеть создать анимацию сочлененных объектов. Это действительно сложная задача. Если помните, мы рассматривали, как можно анимировать объекты при помощи анимации каркаса или анимации подобъектов. В первом случае нам нужна программа трехмерного моделирования, которая выдает кадры с анимированным каркасом, которые вы должны считать. Во втором случае вы должны иметь возможность загрузить каркас с указанием точек сочленения, а затем загрузить информацию о движении (наподобие файлов движения *BioVision*) и осуществить анимацию. Это тоже очень сложная задача, и несмотря на то, что соответствующий инструментарий может дать вам необходимые данные (как об объекте, так и о его движении), вам все равно потребуется написать код для того, чтобы использовать эти данные в вашем игровом процессоре. Однако об этом мы поговорим немного попозже. В качестве примеров такого инструментария можно привести *3D Studio Max*, *Maya* и *trueSpace*.

## Загрузка внешних данных

Как я говорил, мы напишем весь необходимый код, как только в нем возникнет потребность, а пока я просто хочу, чтобы вы прочувствовали ситуацию и познакомились с некоторыми форматами файлов — просто чтобы иметь о них представление. Главная цель любого формата — иметь возможность представить трехмерный объект с достаточным для игрового процессора количеством информации. Сейчас нам не потребуется знание о текстурах, освещении и т.д. — нам нужно только базовое описание объекта в виде списка вершин и сетки многоугольников.

Однако, по мере чтения книги, нам понадобится очень многое! Например, если мы используем программу трехмерного моделирования для создания танка, то нам явно потребуется не только список вершин и сетка многоугольников, но и атрибуты многоугольников, координаты текстур и многое другое, словом, потребуется более мощный формат файла. Кроме того, если программа будет моделировать игровой мир целиком, то необходимо будет хранить информацию, например, об источниках освещения.

Детальное описание всевозможных форматов файлов явно выходит за рамки **данной книги**, а следовательно, и за рамки данной главы, так что здесь мы бегло ознакомимся только с некоторыми наиболее популярными форматами.

## PLG

Формат PLG был создан программистами, разработавшими **REND386** (одним из них был Берни Рол (**Bernie Roehl**), причем лично я думаю, что он и сделал всю эту работу). Формат очень прост и идеально подходит для того, чтобы начать знакомство именно с него. К сожалению, в нем не хватает информации о текстурах и двусторонних многоугольниках. Основная причина использования данного формата состоит в том, что многие программы трехмерного моделирования генерируют файлы в данном формате, а также существуют утилиты для преобразования других форматов, например, **.DXF**, в формат PLG. При этом написание кода анализатора данного формата занимает час-два, а не неделю, необходимую для наиболее сложных современных форматов.

### Заголовок

PLG-файл начинается с имени объекта и количества вершин и граней:

`object_name num_vertices num_polygons`

Например, строка

`pyramid 5 5`

говорит о том, что далее следуют данные объекта `pyramid`, у которого 5 вершин и 5 граней.

### Список вершин

Следующая часть данных представляет собой вершины в порядке от 0 до **n-ой**. Каждая вершина отделяется пробельным символом и представляет собой значения координат в порядке **x-y-z**. Для пирамиды, показанной на рис. 6.51, данные о вершинах **выглядят** следующим образом:

```
0 20 0
30 -10 30
30 -10 -30
-30 -10 -30
-30 -10 30
```

### Список многоугольников

Очередная часть файла является описанием многоугольников, которое **представляет** собой набор строк, **описывающих** цвет многоугольника, количество **вершин** и индексы вершин. Формат описания будет следующим:

`surface_description n v1 v2 v3 ... vn`

где `surface_description` — дескриптор поверхности, о котором мы поговорим немного позже, `n` — количество вершин, а `v1, v2, ..., vn` — список вершин в порядке против часовой стрелки. Запятые в файле не используются.

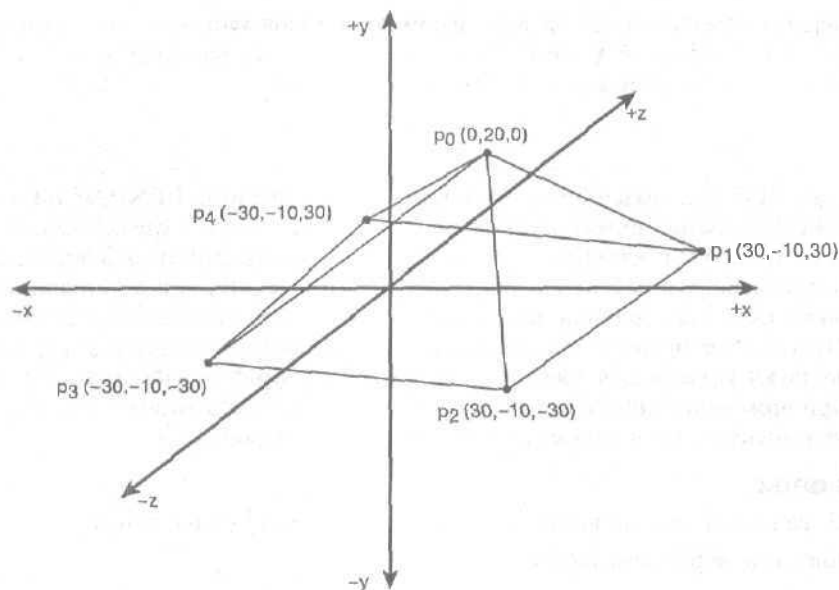


Рис. 6.51. Пятигранная пирамида

Список поверхностей нашей пирамиды выглядит следующим образом.

```
103021
103032
103043
103014
1041234
```

Эти данные означают, что у нас имеется пять многоугольников. Первое число каждой строки называется дескриптором поверхности, следующее определяет количество вершин конкретного многоугольника. Например, многоугольник во второй строке имеет три вершины  $(0,3,2)$ , а многоугольник в пятой строке — четыре вершины  $(1,2,3,4)$ . Мы договаривались о том, что все наши модели будут состоять исключительно из треугольников, но здесь, с иллюстративной целью, мы отошли от этого правила, чтобы показать поддержку форматом многоугольников с произвольным количеством сторон.

Единственной загадкой остается первое число в строке, которое называется дескриптором поверхности и может быть либо десятичным, либо шестнадцатеричным числом (начинающимся с 0x или 0X, за которым следует шестнадцатеричное целое значение). Дескриптор поверхности является 16-битовым значением, интерпретируемым как набор битовых полей следующим образом.

$H_{15} R_{14} S_{13} S_{12} C_{11} C_{10} C_9 C_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0$

- Бит R зарезервирован и должен быть равен 0.
- Если установлен бит H, он указывает на то, что младшие 14 битов представляют собой индекс в карте поверхности.
- Если бит H равен 0, биты SS интерпретируются так, как показано в табл.6.1.

Таблица 6.1. Значение битов S5 дескриптора поверхности

$S_{13}S_{12}$	Определение
00	Данный многоугольник закрашивается определенным цветом без каких-либо специальных эффектов. Если биты CCCC равны 0, то биты BBBBBBBB непосредственно определяют один из 256 цветов в 8-битовом режиме. Ненулевые значения CCCC определяют один из 16 оттенков цвета, а старшие четыре бита BBBBBBBB — уровень яркости
01	Многоугольник имеет плоское затенение, т.е. он должен выводиться с постоянным затенением, которое определяется углом падения освещения. Таким образом, при перемещении многоугольника его яркость будет изменяться. Биты CCCC определяют один из 16 оттенков, а биты BBBBBBBB — "яркость" цвета, которая будет умножаться на косинус угла между нормалью к многоугольнику и вектором от многоугольника к источнику света. Если значение CCCC равно 0, цвет многоугольника — черный
10	Многоугольник рассматривается как "металлический". Биты CCCC (которые должны быть ненулевыми) определяют один из 16 оттенков, а старшие 5 битов BBBBBBBB используются в качестве смещения цикла диапазона яркости (циклическое изменение осуществляется с целью придания металлического эффекта)
11	Многоугольник прозрачный

Табл. 6.1 отражает исходное определение формата PLG, которое несколько устарело, т.к. ориентировано на 4- и 8-битовые цветовые режимы. Хотя мы и работаем с 8-битовым режимом, на самом деле нам нужен 16-битовый режим. Таким образом, нам придется немного подкорректировать формат, когда мы захотим использовать его в следующей главе для поддержки 16-битовых RGB-цветов. Ясно, что программа загрузки не знает о наших усовершенствованиях, но мы можем для наших экспериментов добавить соответствующую поддержку вручную.

### Загрузка объектов

Загрузка PLG-файла не представляет особых трудностей. Это обычный ASCII файл с несколькими строками текста, описывающими объект. Самым сложным при его обработке является чтение дескрипторов, которые могут быть представлены как в десятичном, так и в шестнадцатеричном формате, в зависимости от наличия или отсутствия ведущих символов 0x; кроме того, в файле могут быть комментарии, начинающиеся с; символа #. Сейчас мы не будем заниматься написанием анализатора PLG-файла, но это не сложно — надо просто считывать строки по одной и разбирать имеющиеся в них числа. Это очень удобный формат для экспериментов, и мы, вероятно, воспользуемся им в следующей главе. Напоследок приведу пример полного PLG-файла.

```
pyramid 5 5
```

```
# Список вершин
```

```
0 20 0
30 -10 30
30 -10 -30
-30 -10 -30
-30 -10 30
```

```
# Список многоугольников
```

```
10 3 0 2 1
10 3 0 3 2
```

Вот и все. Теперь вы понимаете, почему мне так нравится этот формат файла? Он прост, с него легко начать изучение, для него легко написать анализатор. Большинство программ трехмерного моделирования не работают с **PLG-форматом** непосредственно из-за его простоты, однако имеется ряд конверторов в данный формат. Надо только не забывать о том, что формат ориентирован на правую систему координат и обход вершин против часовой стрелки, в то время как в нашей книге мы работаем с левой системой координат.

## NFF

Следующий по простоте формат — *нейтральный формат файла* (neutral file format, NFF). Я впервые использовал данный формат много лет назад, и работал он вполне прилично, так что я не вижу препятствий, чтобы воспользоваться им и сейчас. Он был разработан Эриком Хайнсом (Eric Haines). Последняя версия данного формата — 3.1, и именно с ней мы и познакомимся.

Формат NFF разрабатывался для определения, в дополнение к объектам, простых миров. В нем **поддерживаются** такие вещи, как камера, фоновый цвет, положение источников света и т.п. Эти возможности очень слабо влияют на сложность анализа NFF-файлов, так что этот формат хорошо подходит для наших целей.

### Поддерживаемые форматом возможности

Формат NFF поддерживает:

- простую аксонометрическую область обзора;
- описание фонового цвета;
- описание положения источников освещения;
- описание свойств поверхностей;
- описание многоугольников, цилиндров, конусов, сфер.

NFF файлы представляют собой набор строк. Для каждой описываемой сущности первым полем идет описание ее типа. Остальная часть строки (и, возможно, последующие строки) содержат остальную **информацию** о сущности. Типы описываемых **сущностей** перечислены ниже:

- **v** — векторы и углы обзора;
- **b** — цвет фона;
- **I** — положения источников света;
- **f** — свойства материалов объекта;
- **c** — описание цилиндра или конуса;
- **s** — описание сферы;
- **p** — **описание** многоугольника;
- **pp** — описание многоугольного фрагмента.

Рассмотрим вкратце главные типы объектов, с которыми мы будем иметь дело. При рассмотрении я буду выделять ключевые слова полужирным шрифтом, оставляя параметры (**которые**, как правило, представляют собой числа с плавающей точкой) набранными обычным шрифтом.

### Положение точки обзора

Положение точки обзора определяет точку обзора вместе с некоторыми параметрами области обзора (хотя и не всеми). Описание представлено ниже.

```
v
from   Fx Fy Fz
at     Ax Ay Az
up     Ux Uy Uz
angle  angle
hither hither
resolution xres yres
```

Вот что все это означает:

- from — **положение** точки обзора в мировых координатах;
- at — положение целевой точки (центр изображения) в мировых координатах;
- up — вектор, определяющий направление "вверх" относительно вектора at;
- angle — угол обзора в градусах, определяемый от **центра** верхней строки до центра нижней, и от центра левого столбца до центра правого;
- hiter — расстояние до ближней плоскости отсечения (если таковая имеется);
- resolution — разрешение плоскости проекции в пикселях.

#### НА ЗАМЕТКУ

Никаких предположений о нормализации данных не делается. Кроме того, векторы необязательно должны быть перпендикулярны друг другу. Кроме того, по умолчанию дальняя плоскость отсечения **располагается** на бесконечности, а форматное отношение равно 1.0.

По соглашению определение положения точки обзора должно предшествовать определению всех других объектов.

### Фоновый цвет

Фоновый цвет представляет собой обычное RGB-значение, причем значение каждого канала находится в пределах от 0 до 1:

```
b R G B
```

Если фоновый цвет не определен, считается, что он равен  $\{0,0,0\}$ .

### Положение источника света

Источники света определяются посредством указания их координат и (необязательно) цвета:

```
{ X Y Z [R G B]
```

Здесь X, Y, Z — положение источника **света**, а необязательные значения R, G, B определяют его цвет. Все источники света должны быть определены до определения любого из объектов. Учтите, что если цвет источника не указан, то ему будет присвоено некоторое неопределенное ненулевое значение.

### Цвет заполнения и параметры затенения

**Цвет** заполнения и параметры затенения определяют свойства освещения всех объектов, определения которых следуют за данным (до тех пор, пока эти параметры не будут изменены новым определением цвета заполнения и параметров затенения).

```
f red green blue Kd Ks Shine T index_of_refraction
```

Далее перечислены параметры этого описания:

- `red, green, blue` — RGB-значения в диапазоне от 0 до 1;
- `Kd` — диффузный компонент отражения;
- `Ks` — зеркальный компонент отражения;
- `Shine` — степень косинуса для затенения по Фонгу;
- `T` — прозрачность.

Большинство данных параметров пока не несут для нас смысловой нагрузки, так как мы еще не разбирались с освещением объектов, но упрощенно можно сказать, что имеется простая формула, которая определяет отражение источника света от поверхности с учетом рассеянной и зеркальной части отражений, в которой и участвуют перечисленные параметры.

### Определение многоугольника

Многоугольник определяется как множество вершин, которые должны находиться в одной плоскости. В формате NFF все многоугольники односторонни по определению; вершины упорядочиваются против часовой стрелки; используется правая система координат. Кроме того, две первые стороны должны образовывать ненулевой выпуклый угол, с тем чтобы нормаль к многоугольнику и его видимость можно было определить по первым трем вершинам.

#### ВНИМАНИЕ

Не забывайте о том, что формат NFF основан на правой системе координат! Будьте внимательны!

```
p num_vertices  
v1.x v1.y v1.z  
v2.x v2.y v2.z
```

```
vn.x vn.y vn.z
```

Здесь `num_vertices` определяет количество вершин многоугольника, а `vi` представляют каждую вершину.

### Многоугольный фрагмент

Многоугольный фрагмент (`polygonal patch`) определяется набором вершин и их нормалей. Формат практически идентичен определению многоугольника, но с добавлением в виде векторов нормалей в каждой вершине.

```
pp num_vertices  
v1.x v1.y v1.z n1.x n1.y n1.z  
v2.x v2.y v2.z n2.x n2.y n2.z
```

```
vn.x vn.y vn.z nn.x nn.y nn.z
```

Здесь `num_vertices` — количество вершин, а `vi` и `ni` представляют, соответственно, вершины и их нормали.

#### НА ЗАМЕТКУ

Формат NFF имеет ряд других возможностей, таких как описание примитивов наподобие сфер и конусов, однако перечисленные возможности по сути покрывают все наши потребности, за исключением, быть может, символа комментария `#`.

## Пример NFF файла

Для того чтобы завершить знакомство с NFF-файлами, приведем пример файла, в котором описан куб 2x2x2.

```
v
from 02.02.0
at 000
up 010
angle 40
hither 1
resolution 512 512
l0 5 0
l0 2.0 2.0
f0.000000 0.000000 0.000000 1 0 0 0 0
P3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
p3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
p3
-1.000000 1.000000 -1.000000
-1.000000 -1.000000 -1.000000
1.000000 -1.000000 -1.000000
P3
-1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 -1.000000
P3
1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 -1.000000 1.000000
P3
1.000000 1.000000 1.000000
1.000000 -1.000000 1.000000
-1.000000 -1.000000 1.000000
p3
1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000
-1.000000 1.000000 1.000000
P3
-1.000000 1.000000 -1.000000
1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
P3
-1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
```

```

-1.000000 1.000000 1.000000
P3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 -1.000000
-1.000000 1.000000 1.000000
P3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000

```

## 3D Studio

3D Studio и его потомок, 3D Studio Max (или просто Max) — одни из наиболее широко используемых программ трехмерного моделирования в мире, как в области создания игр, так и в области кинематографа. Сам комплекс 3D Studio ведет свою родословную от еще более старой программы — AutoCad производства AutoDesk. AutoCad первоначально работал с форматом DXF, однако с развитием 3D Studio появился новый формат — 3DS, а затем и новый формат MAX, который включает как сетку каждого объекта, так и все данные сцены в целом. К сожалению, это весьма сложные форматы, и могут быть как в текстовом, так и в бинарном виде. Соответственно, я не собираюсь давать здесь детальное описание этого формата, как было в случае форматов PLG и NFF, просто из-за ограниченного объема книги. Поэтому я дам только минимальное описание форматов и приведу очень небольшие примеры. Однако позже мы, возможно, вернемся к вопросам поддержки DXF и/или 3DS.

## DXF

Файлы DXF (Data eXchange Format, формат обмена данными) изначально поддерживались программой AutoCad. Имеются как бинарная, так и текстовая версия данного формата, причем обе имеют одно и то же имя. Формат на самом деле неплох, но он никогда не преследовал цель представления трехмерных моделей освещения, текстур и т.п. — это связано с тем, что он предназначался для инженерных разработок, а не для игр. В итоге поддержку трехмерных моделей в данном формате нельзя назвать полной. Кроме того, в DXF-файлах при представлении любых объектов оказывается много технической информации, не имеющей прямого отношения к самим объектам, так что при анализе данных файлов мы вынуждены разбирать и горы "мусора".

### Формат файла

Стандартный текстовый DXF-файл состоит из следующих основных разделов.

- Заголовок (HEADER) — содержит общую информацию о модели.
- Таблицы (TABLES) — содержит определение именованных элементов.
- Блоки (BLOCKS) — содержит определение *блоков*, описывающих элементы, составляющие блоки чертежа.
- Сущности (ENTITIES) — содержит элементы, описывающие модель наряду с блоками.
- Маркер конца файла (END OF FILE) — каждый DXF-файл завершается строкой EOF.

В каждом разделе могут находиться любые типы данных. Эти данные группируются в классы с т.н. кодами групп. Эти коды групп помогают проводить анализ содержимого файла, так как код позволяет быстро определить контекст данных. В табл. 6.2 приведены основные коды групп.

**Таблица 6.2. Классы кодов групп DXF-файла**

Диапазон кода группы	Означает
0–9	Строковое значение
10–59, 210–239	Значение с плавающей точкой
60–79	Целое значение
999	Комментарий

Что бы ни находилось в DXF-файле, первым всегда будет код группы. Таким образом, сначала вы считываете код группы (который, как правило, представляет собой излишний расход памяти, но облегчает написание анализаторов), после чего вы можете сразу определить, какого типа параметр должен считываться за ним. Более детальная расшифровка кодов групп приведена в табл. 6.3, но учтите — в 99% случаев эти значения нас интересовать не будут.

**Таблица 6.3. Детальное описание кодов групп DXF-файлов**

Код группы	Означает
0	Определяет начало файла, описания объекта или таблицы
1	Первичное текстовое значение в описании объекта
2	Некоторое имя
3–4	Прочая текстовая информация
6	Тип линии
7	Стиль линии
8	Имя слоя
9	Идентификатор имени переменной, используется в разделе HEADER
10	Основная координата x
11–18	Дополнительные координаты x
20	Основная координата y
21–28	Дополнительные координаты y
30	Основная координата z
31–37	Дополнительные координаты z
38	Высота
39	Толщина
40–48	Общие значения с плавающей точкой
49	Повторное значение
50–58	Углы
62	Номер цвета
66	Флаг "следует объект"
70–78	Целые значения
210, 220, 230	Следуют координаты X, Y, Z экструзии
999	Комментарии

В файле всегда сначала идет код группы, а уже потом — настоящая информация. Например, в начале каждого DXF-файла находится строковое значение, указывающее на начало файла. В соответствии с табл. 6.2 и табл. 6.3, строковые значения имеют код группы 0–10, а начало файла — код 0, так что в начале каждого DXF-файла вы можете увидеть нечто подобное следующему:

```
0
SECTION
```

Предположим, что вы хотите добавить комментарий — это можно сделать при помощи кода 999 примерно следующим образом:

```
999
Некий комментарий к файлу
```

Рассмотрим теперь обобщенный пустой DXF-файл целиком. Комментарии в стиле C++ (//) добавлены мною в иллюстративных целях и в сам DXF-файл не входят!

```
0 // Начало раздела HEADER
SECTION

2 // Код группы
HEADER

0 // Код группы
ENDSEC // Конец раздела HEADER

0 // Начало раздела TABLES
SECTION

2 // Код группы TABLES
TABLES

0 // Код группы TABLE
TABLE

2 // Окрашен обзор
VPORT
70
// Максимальное количество записей в таблице
// Информация об отдельных элементах располагается
// здесь >>>>

0 // Конец TABLE
ENDTAB

0 // TABLE
TABLE

2 // Код группы позволяет приводить описания LTYPE,
// LAYER, STYLE, VIEW, UCS или DWGMGR
70
// Максимальное количество элементов в таблице
// Информация об отдельных элементах располагается
// здесь >>>>
```

```

0 // Конец TABLE
ENDTAB

0
ENDSEC // Конец раздела TABLES

0 // Начало раздела BLOCKS
SECTION

2 // BLOCKS
BLOCKS

    // Определения блоков

0
ENDSEC // Конец раздела BLOCKS

0 // Начало раздела ENTITIES
SECTION

2 // Здесь начинаются описания объектов
ENTITIES

    // Информация об объектах

0
ENDSEC // Завершение раздела ENTITIES

0
EOF // Конец файла

```

Как видите, даже в пустом файле записано немало. К счастью, все, что нас интересует, находится в одном разделе — ENTITIES, поскольку именно здесь располагаются все трехмерные данные. Есть целый ряд описываемых объектов, которые перечислены в табл. 6.4.

**Таблица 6.4. Некоторые типы объектов DXF-файла**

<i>Дескриптор</i>	<i>Описание</i>
LINE	Прямая
POINT	Точка
CIRCLE	Окружность
ARC	Дуга
TRACE	Луч в пространстве
SOLID	Сплошной объект
TEXT	Текстовая строка
SHAPE	Форма
POLYLINE	Набор соединенных отрезков
VERTEX	Отдельная вершина
3DFACE	Трехмерная поверхность или многоугольник

По сути, нас интересует только тип 3DFACE. Заметим, что здесь нет ни источников освещения, ни текстур или камер. DXF — простейший формат описания каркасов и не более того. Рассмотрим **описание** в DXF-файле четырехугольника (здесь вновь комментарии // добавлены мною и не являются частью файла!).

O // Начало объекта, на что указывает код O  
3DFACE

10 // Компонент x вершины 0 четырехугольника  
x0 // и его значение

20 // Компонент y вершины 0 четырехугольника  
y0 // и его значение

30 // Компонент z вершины 0 четырехугольника  
z0 // и его значение

// Для очередной вершины увеличиваем коды групп XYZ на 1

11 // Компонент x вершины 1 четырехугольника  
x1 // и его значение

21 // Компонент y вершины 1 четырехугольника  
y1 // и его значение

31 // Компонент z вершины 1 четырехугольника  
z1 // и его значение

// Для очередной вершины увеличиваем коды групп XYZ на 1

12 // Компонент x вершины 2 четырехугольника  
x2 // и его значение

22 // Компонент y вершины 2 четырехугольника  
y2 // и его значение

33 // Компонент z вершины 2 четырехугольника  
z2 // и его значение

// Для очередной вершины увеличиваем коды групп XYZ на 1

13 // Компонент x вершины 3 четырехугольника  
x3 // и его значение

23 // Компонент y вершины 3 четырехугольника  
y3 // и его значение

33 // Компонент z вершины 3 четырехугольника  
z3 // и его значение

НА ЗАМЕТКУ

Если вы хотите **определить** треугольник, то последняя вершина v3 должна иметь те же значения, что и v2 (т.е. вершины должны быть v0, v1, v2, v2).

А вот как выглядит в формате DXF наш рассмотренный ранее куб (здесь для экономии места представлено несколько столбцов, хотя в настоящем файле каждое значение записывается на отдельной строке).

0	0	0	0
SECTION	SECTION	3DFACE	3DFACE
2	2	8	8
HEADER	HEADER	CUBE	CUBE
0	0	10	10
ENDSEC	ENDSEC	-1.000000	1.000000
0	0	20	20
SECTION	SECTION	1.000000	1.000000
2	2	30	30
TABLES	TABLES	0.000000	2.000000
0	0	11	11
TABLE	TABLE	-1.000000	1.000000
2	2	21	21
LAYER	LAYER	-1.000000	-1.000000
70	70	31	31
153	153	0.000000	2.000000
0	0	12	12
LAYER	LAYER	1.000000	-1.000000
2	2	22	22
Cube	Cube	-1.000000	-1.000000
70	70	32	32
62	0	0.000000	2.000000
62	62	13	13
15	15	1.000000	-1.000000
6	6	23	23
CONTINUOUS	CONTINUOUS	1.000000	1.000000
0	0	33	33
ENDTAB	ENDTAB	0.000000	2.000000
0	0	62	62
ENDSEC	ENDSEC	0	0
0	0	0	0
SECTION	SECTION	3DFACE	3DFACE
2	2	8	8
ENTITIES	ENTITIES	CUBE	CUBE
0	0	10	10
3DFACE	3DFACE	1.000000	-1.000000
8	8	20	20
CUBE	CUBE	1.000000	1.000000
10	10	30	30
1.000000	1.000000	0.000000	0.000000
20	20	11	11
-1.000000	-1.000000	1.000000	1.000000
30	30	21	21
0.000000	0.000000	-1.000000	1.000000
11	11	31	31
-1.000000	-1.000000	0.000000	0.000000
21	21	12	12
-1.000000	-1.000000	1.000000	1.000000
31	31	22	22

0.000000	0.000000	-1.000000	1.000000
12	12	32	32
-1.000000	-1.000000	2.000000	2.000000
22	22	13	13
-1.000000	-1.000000	1.000000	-1.000000
32	32	23	23
2.000000	2.000000	1.000000	1.000000
13	13	33	33
1.000000	1.000000	2.000000	2.000000
23	23	62	62
-1.000000	-1.000000	0	0
33	33		
2.000000	2.000000		
62	62		
0	0		

#### НА ЗАМЕТКУ

Обратите внимание на 8 с последующим CUBE после каждого определения 3DFACE. Это вполне корректное определение. В данном случае 8 — код группы имени слоя (layer name), и строка CUBE просто определяет это имя. Слой в данном случае — это имя (CUBE) трехмерного экспортируемого объекта.

Как видите, анализ данного файла не так сложен, как могло показаться сначала. Вы просто находите раздел ENTITIES и считываете все объекты 3DFACE. Единственная неприятность заключается в том, что DXF-файлы не хранят списки вершин отдельно — для каждой поверхности используются независимые вершины.

### 3DS/ASCII

3D Studio Max поддерживает ряд различных форматов, однако "родными" форматами являются форматы 3DS и ASCII. Это по сути один и тот же формат, но 3DS — формат бинарный, а ASCII — текстовый. Этих форматов достаточно практически для всего, что нам может потребоваться, но в силу их сложности я не буду пояснять их. Я думаю, что если вы взглянете на дампы файла с описанием куба 2x2x2, расположенного в точке (0,0,0), то вы увидите, как работает данный формат.

Ambient Light color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Cube"

Tri-mesh, Vertices: 8 Faces: 12

Vertex List:

Vertex 0: X:-1.000000 Y:-1.000000 Z:-1.000000

Vertex 1: X:-1.000000 Y:-1.000000 Z:1.000000

Vertex 2: X:1.000000 Y:-1.000000 Z:-1.000000

Vertex 3: X:1.000000 Y:-1.000000 Z:1.000000

Vertex 4: X:-1.000000 Y:1.000000 Z:-1.000000

Vertex 5: X:1.000000 Y:1.000000 Z:-1.000000

Vertex 6: X:1.000000 Y:1.000000 Z:1.000000

Vertex 7: X:-1.000000 Y:1.000000 Z:1.000000

Face list:

Face 0: A:2 B:3 C:1 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 1: A:2 B:1 C:0 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1  
 Face 2: A:4 B:5 C:2 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 3: A:4 B:2 C:0 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 4: A:6 B:3 C:2 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 5: A:6 B:2 C:5 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 6: A:6 B:7 C:1 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 7: A:6 B:1 C:3 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 8: A:6 B:5 C:4 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 9: A:6 B:4 C:7 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 10: A:1 B:7 C:4 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1  
 Face 11: A:1 B:4 C:0 AB:1 BC:1 CA:1  
 Material:"r255g255b255a0"  
 Smoothing: 1

## COB-файлы Caligari

Постепенно, приобретая опыт, я стал горячим поклонником и пропагандистом Caligari trueSpace. Это очень мощная профамма трехмерного моделирования, обладающая очень простым и понятным интерфейсом. Формат COB похож на формат 3DS и также имеет бинарную и текстовую версии. Этот формат также очень сложен для того, чтобы детально его описывать, так что я опять приведу дамп файла с описанием уже знакомого нам куба.

```
Caligari V00.01ALH
PolH V0.06 Id 16796788 Parent 0 Size 00000981
Name Cube
center 0 0 1
x axis 1 0 0
y axis 0 1 0
z axis 0 0 1
Transform
1 0 0
0 1 0
0 0 1
0 0 1
WorldVertices 8
```

```

-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 -1.000000
1.000000 -1.000000 1.000000
-1.000000 1.000000 -1.000000
1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000

```

Texture Vertices 14

```

0.000000 0.333333
0.000000 0.666667
0.250000 0.333333
0.250000 0.666667
0.500000 0.000000
0.500000 0.333333
0.500000 0.666667
0.500000 1.000000
0.250000 0.000000
0.250000 1.000000
0.750000 0.333333
0.750000 0.666667
1.000000 0.333333
1.000000 0.666667

```

Faces 6

Face verts 4 flags 0 mat 0

```
<2,2> <0,0> <1,1> <3,3>
```

Face verts 4 flags 0 mat 0

```
<4,4> <0,8> <2,2> <5,5>
```

Face verts 4 flags 0 mat 0

```
<5,5> <2,2> <3,3> <6,6>
```

Face verts 4 flags 0 mat 0

```
<6,6> <3,3> <1,9> <7,7>
```

Face verts 4 flags 0 mat 0

```
<4,10> <5,5> <6,6> <7,11>
```

Face verts 4 flags 0 mat 0

```
<0,12> <4,10> <7,11> <1,13>
```

DrawFlags 0

Radiosity Quality: 0

Unit V0.01 Id 16796789 Parent 16796788 Size 00000009

Units 2

Mat1 V0.06 Id 16799524 Parent 16796788 Size 00000090

mat# 0

shatter: phong facet: auto32

rgb 1,1,1

alpha 1 ka 0.23 ks 0.81 exp 0.91 ior 1

ShBx V0.00 Id 16799525 Parent 16799524 Size 00000627

Shader class: color

Shader name: "chrome" (chrome)

Number of parameters: 3

base colour: color (255, 255, 255)

vector: vector (0, 1, 0.5)

mix: float 0.5

Shader class: transparency

```

Shader name: "none" (none)
Number of parameters: 0
Shader class: reflectance
Shader name: "caligari phong" (caligari phong)
Number of parameters: 8
ambient factor: float 0.23
diffuse factor: float 0.19
specular factor: float 0.81
exponent: float 43.8135
specular colour: color (61, 67, 255)
transmission factor: float 0
mirror factor: float 0.78084
refraction: float 1
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
END V1.00 Id 0 Parent 0 Size 0

```

Как видите, это еще один формат, со списком поверхностей, **базирующимся** на описании вершин. Формат вполне приемлем для восприятия и позволяет разобраться в описании различных объектов, в том числе источников освещения, текстур и т.п.

Кроме того, **trueSpace** использует еще один формат — **.SCN**, который поддерживает целые сцены и миры. 3D Studio Max также поддерживает эти возможности в формате **.MAX**. Таким образом, потенциально мы можем использовать любую программу трехмерного моделирования из перечисленных не только для определения объектов, но и для моделирования целых миров с источниками освещения. Таким образом, использование программ трехмерного моделирования (которые разработчики делают с каждым днем все мощнее и все проще в обращении) может существенно облегчить вам главную задачу — написание трехмерной игры.

#### НА ЗАМЕТКУ

3D Studio Max имеет встроенный язык сценариев, который называется **MaxScript**. С его помощью вы можете получить доступ ко всем объектам трехмерного мира, так что в принципе, написав соответствующие сценарии и макросы, можно даже превратить 3D Studio Max в игровой редактор!

## .X-файлы Microsoft DirectX

Есть еще один формат, о котором бы хотелось вкратце упомянуть, — это формат **Microsoft DirectX**. Я отнюдь не эксперт в этом формате и никогда особенно им не пользовался, кроме как для импорта информации при помощи **Direct3D**. Таким образом, я никогда не писал анализатора для данного формата файлов. Могу только сказать, что это очень сложный, хотя и очень мощный формат, так что, пожалуй, даже не будем мечтать о том, чтобы **самостоятельно** писать анализатор для него. Впрочем, если вы хотите познакомиться с этим форматом поближе, то на прилагаемом компакт-диске вы найдете соответствующие материалы.

## Краткое резюме

Как видите, различные форматы файлов для хранения трехмерных данных имеют свои "за" и "против", но при этом удобочитаемость (наличие ASCII-версий) определенно является существенным плюсом. Если вас интересуют и другие форматы файлов, воспользуйтесь книгой Кейт Рал (Keith Rule) *3D Graphics File Formats*, изданную Addison-Wesley, либо поищите соответствующую информацию в Internet. Вот пара адресов, по которым вы сможете найти списки описаний форматов файлов:

НА ЗАМЕТКУ

Главное, о чем следует помнить, — все рассмотренные нами форматы экспортируют координаты в правой системе координат. Я хочу еще раз обратить ваше внимание на этот факт. При написании анализаторов нам надо будет внести в код соответствующие коррективы, чтобы данные были правильно перенесены в используемую в данной книге левую систему координат.

## Основы преобразований твердых тел и анимации

Надо полагать, что если вы дочитали книгу до этого места, то у вас имеется неплохая подготовка в области технологий двумерной анимации, таких как перенос, масштабирование и вращение. Однако с объектами в трехмерном мире мы **можем** выполнить гораздо больше действий, хотя бы потому, что у нас есть шесть степеней свободы; кроме того, возможно выполнение **деформации** самого каркаса объекта.

Мы уже упоминали об анимации на основе кадров и на основе данных о движении; оба этих варианта будут реализованы далее в книге, когда мы будем говорить об анимации персонажей. Сейчас же я хочу только бегло пройти по азам анимации для простого перемещения трехмерных объектов и изменения их ориентации.

### Трехмерное перемещение

**Перемещение** в трехмерном пространстве реализуется тривиально, и мы уже осуществляли его — как вручную, так и при помощи матричных операций, так что мне не надо тратить время на описание этого процесса. Я хочу только сказать несколько слов о перемещении каркаса в целом, вернее — о наилучшем способе **перемещения** каркасов в трехмерных играх.

Поскольку каждый объект имеет множество локальных координат, указываемых относительно начала локальных координат  $(0,0,0)$ , мы не должны изменять локальные координаты объекта для того, чтобы переместить объект. Лучшим подходом будет определение положения объекта с последующим использованием его в качестве точки отсчета при преобразовании локальных координат в мировые. У нас уже имеется поле для отслеживания положения объекта — `world_pos`. Таким образом, мы откладываем **реальный** перенос каждой точки каркаса до стадии преобразования локальных координат в мировые, и таким образом для перемещения всего объекта в целом нам надо **переместить только одну точку**. Итак, если вы хотите переместить объект на вектор  $t = \langle tx, ty, tz \rangle$ , все, что надо для этого сделать, — это выполнить следующий код:

```
object.world_pos.x+=t.x;  
object.world_pos.y+=t.y;  
object.world_pos.z+=t.z;
```

Относительно просто. Теперь, когда мы знаем о том, что все, что нам надо сделать, — это перенести одну точку, чем мы должны заняться далее? Однако этот вопрос надо адресовать скорее системе искусственного интеллекта. Именно эта система определяет движение объекта. А сейчас давайте просто осуществим **движение** объекта по окружности в трехмерном пространстве. Все, что нам надо для этого, — это вычислить абсолютные координаты объекта в разные моменты времени (или относительные, относительно положения объекта в предыдущий момент времени). Для простоты воспользуемся **абсолют-**

ными координатами. Итак, будем моделировать эллиптическое движение в плоскости xz. Ниже представлен код, решающий поставленную задачу:

```
float x_pos = a*cos(angle);  
float y_pos = altitude;  
float z_pos = b*sin(angle);
```

Вы просто должны подставить соответствующие значения вместо a, b, angle и altitude и получить искомое движение. Слишком просто? Давайте усложним задачу и будем двигать объект по спирали.

```
// Начальные условия
```

```
float altitude = 0;
```

```
float rate = .1;
```

```
// Этот код выполняется на каждом цикле
```

```
float x_pos = a*cos(angle);
```

```
float y_pos = altitude+rate;
```

```
float z_pos = b*sin(angle);
```

Как видите, ничего принципиально сложного, по сути, ничуть не сложнее, чем движение в двумерном случае.

## Трехмерное вращение

Вращение немного сложнее, чем перемещение. Математика не так сложна (хотя и сложнее, чем при перемещении), существенно более сложной задачей является отслеживание ориентации объекта. Например, когда мы перемещаем объект в точку  $(x, y, z)$ , то мы знаем, что объект находится в этой точке, и больше нам ничего не надо знать. Но когда мы поворачиваем объект на угол  $(\alpha_x, \alpha_y, \alpha_z)$ , то это не значит, что ориентация объекта определяется этими углами относительно осей — вспомните о блокировке подвески. Кроме того, при повороте объекта имеет значение порядок поворота — так, поворот XYZ отличается от поворота YXZ. Конечно, если вы вращаете какой-нибудь астероид, создав определенную скорость вращения  $(r_x, r_y, r_z)$ , то вас вряд ли должны волновать эти вопросы. Однако при повороте танка или корабля вы должны иметь возможность в любой момент вычислить его ориентацию в пространстве.

Для поворота объекта в трехмерном пространстве всегда выполняется поворот локальных координат объекта. Поворот выполняется относительно начала локальных координат  $(0, 0, 0)$  (но не мировых — иначе это будет не поворот объекта, а вращение его вокруг начала мировых координат). Здесь есть два подхода. Можно выполнить поворот и затем сохранить новые локальные координаты объекта, а можно сделать текущую ориентацию объекта частью визуализации, т.е. когда объект готов для визуализации, выполняется его поворот и он передается во временный массив. Плюс последнего подхода в том, что при этом не происходит искажения исходного объекта, и он может использоваться как эталонная копия. Недостатком является то, что если поворот разовый, то одни и те же вычисления должны производиться для каждого кадра игры, даже если не выполняется никакая анимация объекта.

С учетом этого давайте все же при повороте сохранять преобразованные локальные координаты объекта. Пусть, например, у вас есть танковый имитатор, и объект `tank1`, который уже определен, загружен и ориентирован вдоль положительного направления оси z. При повороте танка вправо или влево можно поворачивать модель в локальных координатах следующим образом.

```

// Инициализация. Мы считаем, что танк движется в плоскости
// xz, так что поворот выполняется параллельно оси y
// (рыскание)
tank1.dir.x = 0;
tank1.dir.y = 0;
tank1.dir.z = 0;

// Код игры
if (turn_right)
{
    for (int index=0; index<tank1.num_vertices; index++)
    {
        // Предполагается, что данный вызов осуществляет
        // необходимый поворот точки
        Rotate_Point_On_Y(&tank1.vlist_local[index], -5);
        // Обновление текущей ориентации, так что всегда
        // известно, куда направлен танк
        tank1.dir.y-=5;
    } // for
} // if
else if (turn_left)
{
    for (int index=0; index<tank1.num_vertices; index++)
    {
        // Предполагается, что данный вызов осуществляет
        // необходимый поворот точки
        Rotate_Point_On_Y(&tank1.vlist_local[index], 5);
        // Обновление текущей ориентации, так что всегда
        // известно, куда направлен танк
        tank1.dir.y+=5;
    } // for
} // if

```

Код очень прост. Когда игрок **поворачивает влево**, весь список вершин поворачивается параллельно оси  $y$  на  $+5^\circ$ , при повороте вправо — на  $-5^\circ$ . Проблема в том, что по каркасу объекта после поворота вы не можете указать его **направление**, и оно должно вычисляться и **сохраняться** отдельно — что в нашем случае делается при **помощи** поля `tank1.dir`. В любой момент **игры** мы можем проверить это поле и определить, куда же именно направлен танк.

Этот метод работает вполне удовлетворительно, но если начать выполнять повороты по всем трем осям одновременно, то можно попасть в ловушку. Как я уже говорил, реальная ориентация объекта может не быть суммой отдельных поворотов параллельно осям. Однако имеется и лучший путь отслеживания поворотов — **использование вектора ориентации**. Для каждого объекта в нашем трехмерном пространстве при его загрузке и ориентации вдоль положительного направления оси  $z$  вы инициализируете единичный вектор, который также указывает вдоль положительного **направления** оси  $z$ . При любых преобразованиях трехмерного объекта те же преобразования **выполняются** и с этим **единичным вектором** (рис. 6.52). После миллиона поворотов вы просто смотрите, куда теперь указывает вектор, и, таким образом, всегда знаете текущую ориентацию объекта. Именно для этого и используется поле `dir`. Но и здесь есть ловушка. Что если вы поворачиваете объект вокруг оси  $z$ ? Вектор направления вроде бы остается корректным, но **информация** о данном повороте оказывается утрачена, поскольку вектор не в состоянии отслеживать крен (вектор, но не кватернион!).

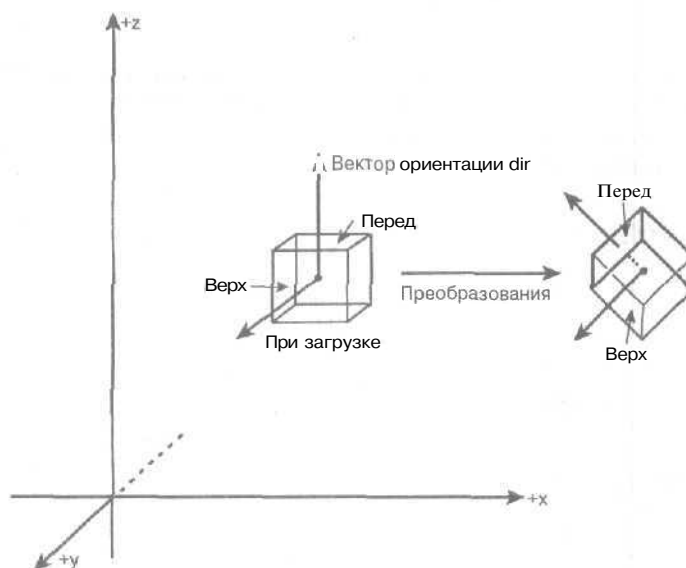


Рис. 6.52. Отслеживание ориентации объекта после загрузки и преобразований

Таким образом, для отслеживания абсолютной ориентации объекта при помощи вектора вам нужен не один, а три вектора в вашей структуре данных (рис. 6.53). Первоначально эти вектора инициализируются локальными осями  $x$ ,  $y$ ,  $z$  объекта, после чего при повороте объекта поворачиваются и упомянутые векторы. После этого ориентацию объекта в любой момент легко вычислить по этим трем векторам, поскольку они подвергаются тем же преобразованиям, что и сам объект.

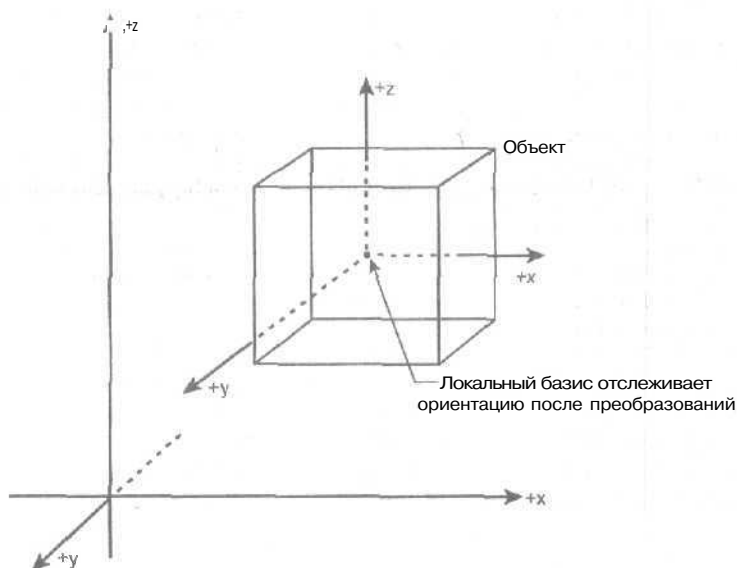


Рис. 6.53. Для полного контроля над ориентацией объекта нужен полный локальный базис

## Морфинг

Последний упрощенный тип анимации — морфинг. Я имею в виду не преобразование одной сетки в другую, а масштабирование и сдвиг сетки в реальном времени для имитации активности объекта. Например, можно записать следующий код для того, чтобы объект выглядел "дышащим".

```
int up_down = 1; // -1 означает уменьшение,
                // 1 означает увеличение

float scale = 1; // Текущий масштаб

while(1)
{
    // Увеличение или уменьшение?
    if (up_down==1)
    {
        Scale_Object(&object, 1.05);
        scale*=1.05;
    } // scale
    else
    {
        Scale_Object(&object, .95);
        scale*=.95;
    } // else
    // Проверка завершения цикла и изменения направления
    // преобразования объекта
    if (scale < .75 || scale > 1.25)
        up_down=-up_down;
} // while
```

Данный код циклически масштабирует объект с изменением в каждом цикле, равным 5%, что дает грубую имитацию дыхания. Ясно, что мы должны написать соответствующую функцию `Scale_Object()`, но это не более чем итерация по передаваемому функции списку вершин и умножение их координат на **масштабирующий** множитель. Главное, на что следует обратить внимание, — это то, что в каждом **цикле** вычислений мы изменяем локальные координаты объекта. В этом нет ничего страшного, но при этом накопление малых ошибок математических вычислений приводит к искажению исходной модели. Это же замечание относится и к поворотам.

### ВНИМАНИЕ

Всегда, когда изменяются локальные координаты, существует **опасность**, что после ряда преобразований объекта из-за накопления ошибок вычислений будут утрачены его исходные данные.

Единственное реальное решение данной проблемы состоит в том, чтобы или хранить эталонную копию **объекта** и почаще обновлять **все** объекты, или выполнять повороты и морфинг "на лету", во время работы конвейера визуализации, при этом используя исходные локальные координаты объекта и сохраняя полученные результаты во временном массиве, не затрагивая исходные данные.

## Обзор конвейера визуализации

Теперь нам необходимо — уже в который раз на протяжении одной главы! — уточнить вид конвейера визуализации с учетом всего сказанного.

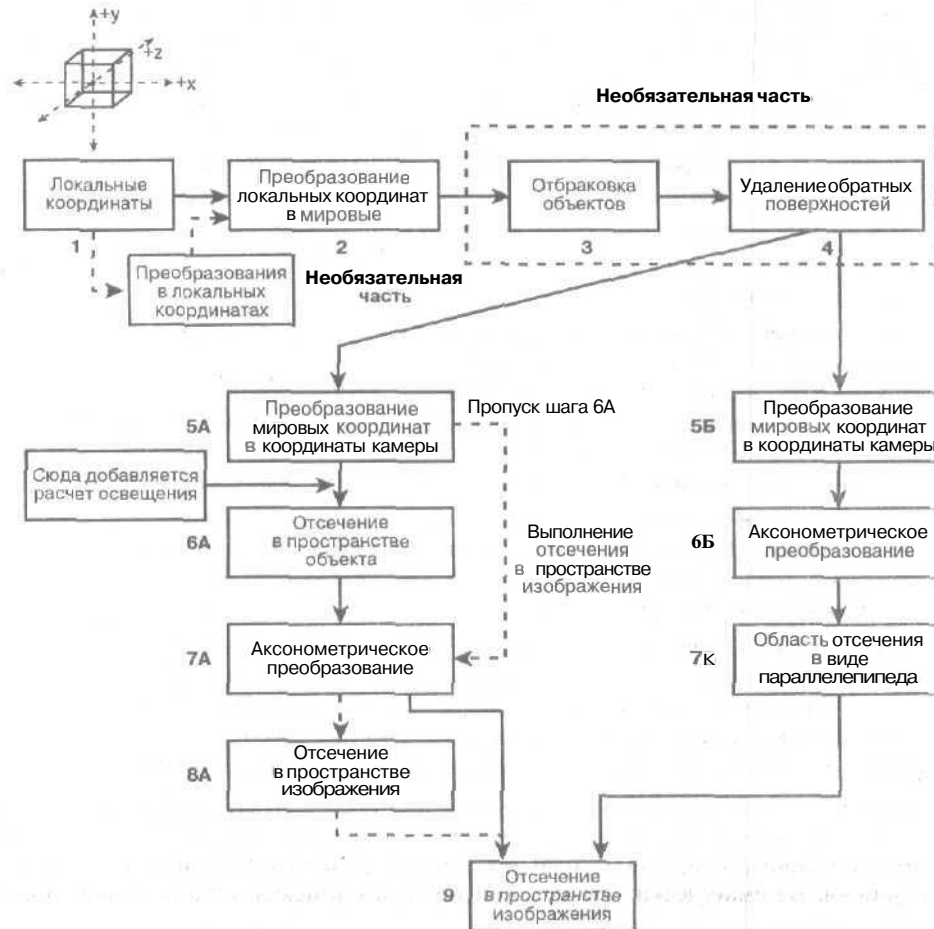


Рис. 6.54. Окончательный вид конвейера визуализации

На рис. 6.54 представлена детальная схема трехмерного конвейера визуализации. Вот ее основные составные части.

- **Локальные преобразования и анимация.** На этом этапе происходит преобразование локальных координат каждого объекта. Это могут быть повороты, масштабирование или другие операции с вершинами, выполняемые в локальной системе координат.
- **Преобразование локальных координат в мировые.** На этом этапе на основе информации о положении объектов в мировой системе координат выполняется преобразование локальных координат объекта в мировые координаты. Информация о положении объекта отслеживается относительно его локального центра при помощи поля `world_pos` (в нашем определении структуры объекта).

- Удаление скрытых поверхностей и освещение. На этом этапе происходит удаление задних поверхностей трехмерных моделей (наряду с отбраковкой ряда полностью невидимых объектов). Кроме того, обычно вычисления освещения также производятся в мировых координатах.
- Удаление объектов. Иногда данный этап выполняется отдельно от других, и может не быть связан с вычислением освещения. Единственная причина, по которой **освещение** и удаление скрытых поверхностей выполняются одновременно, заключается в том, что вычисления нормалей к поверхностям, необходимые для удаления скрытых поверхностей, могут использоваться и моделями освещения.
- Преобразования мировых координат в координаты камеры. Это этап, на котором происходит преобразование потенциально видимой геометрии объектов в систему координат камеры, с учетом ее положения и углов обзора.
- Трехмерное отсечение. На этом этапе выполняется отсечение многоугольников трехмерной областью обзора.
- Аксонометрическое преобразование и отсечение. Здесь происходит преобразование координат в системе отсчета камеры, после которого в картину добавляется перспектива. Кроме того, аксонометрическое преобразование может нормализовать область обзора, превращая ее в куб или прямоугольный параллелепипед, что облегчает решение задачи трехмерного отсечения. Поэтому зачастую трехмерное отсечение выполняется после аксонометрического преобразования.
- Проецирование на экран. Это почти последняя операция в трехмерном конвейере визуализации. При выполнении этого этапа координаты на плоскости проекции отображаются на экранные координаты и передаются подпрограмме растеризации для окончательной визуализации.
- Растеризация и отсечение. Это последний этап трехмерного конвейера визуализации, в который передаются список многоугольников, цвета вершин, карты текстур и т.п., и который **осуществляет** их вывод на экран. Кроме того, некоторые многоугольники могут выходить за рамки экрана (если они не были отсечены на предыдущих стадиях), и будут отсечены **на** этом этапе на уровне пикселей или строк сканирования.

Не так плохо для первоначального рассмотрения конвейера визуализации. Понятно, что в реальном **работающем** конвейере имеется множество деталей, обеспечивающих корректную работу всех составляющих его подпрограмм, но основные составляющие нами рассмотрены. Здесь, правда, по сути ничего не сказано об освещении, но вы можете и сами представить, что это такое — множество математики, работающей с источниками света и ориентацией многоугольников, но об этом мы поговорим в следующих главах.

## Типы трехмерных игровых процессоров

Теперь я хочу ненадолго отвлечься и поговорить о типах трехмерных игровых процессоров. Я не хочу углубляться в используемые ими технологии, а лишь дать вам некоторое представление о них — почему и зачем они существуют и для решения каких задач подходят лучше всего.

### Космические процессоры

На рис. 6.55 показана копия экрана доброй старой космической игры *Star Wars: X-Wing*. Как видите, главная цель игры — сделать объекты летающими в космическом пространстве.

Игры такого типа по ряду причин, как правило, написать гораздо **проще**, чем трехмерные "стрелялки". Так, одной из причин являются более простые анимация, обнаружение столкновений и, например, вопросы баз данных объектов. По большинству **специализированных** вопросов написания трехмерных **игр**, космические игры оказываются **существенно** более простыми. Кроме того, как правило, такие игры работают на уровне объектов, а не многоугольников, т.е. основной единицей работы, удаления, визуализации является объект целиком. Соответственно, на этапе визуализации для сортировки многоугольников, составляющих объект, используется простейший алгоритм художника, или Z-буфер. Объекты легко **сортируются**, поскольку все они представляют корабли в космическом пространстве.

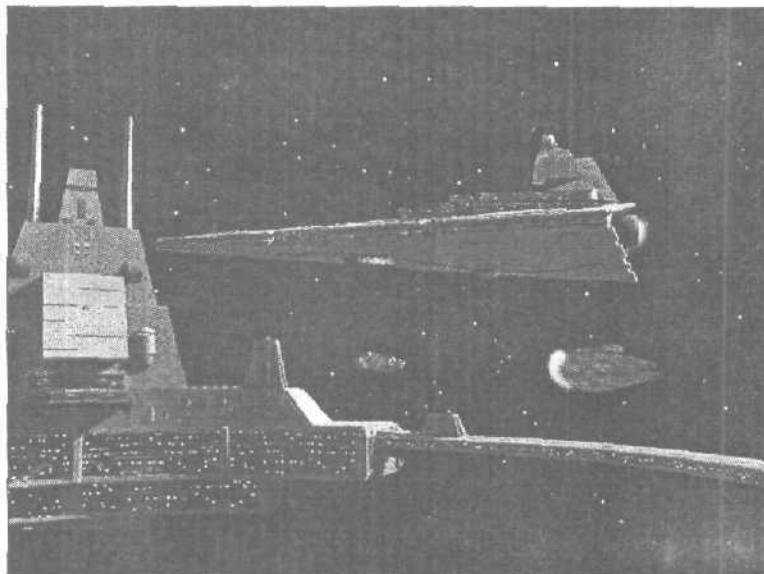


Рис. 6.55. Копия экрана игры Star Wars: X-Wing

Большинство современных космических игр написано с использованием **процессоров**, работающих с многоугольниками. Однако ряд первых трехмерных космических игр типа *Wing Commander* были написаны с использованием процессора **спрайтов**, использующим множество предварительно **визуализированных** видов, а также алгоритмы быстрого отражения и масштабирования растровых изображений. Положения кораблей и т.п. величины представлены отдельными точками в трехмерном пространстве, так что по смыслу процессор можно **считать** трехмерным, в то время как модели таковыми не являлись. Как бы то ни было, современные космические игры **написаны** в основном с использованием процессоров, работающих с многоугольниками, а также с **освещением**, анимацией и т.п. **эффектами**.

Наибольшую сложность в такого рода играх представляет движение. В "стрелялках" очень сложно анимировать "двуногих", но тут спасают предварительно записанные файлы движений, и к тому же перемещение **осуществляется** в основном по двумерной поверхности, например, по полу. В трехмерных космических играх визуализации меньше, но сложнее сами **перемещения** и используемые системы искусственного интеллекта. Это настоящий вызов для программиста, поскольку космический корабль имеет множество степеней свободы, оснащен подвижными орудийными башнями, которые также не **упрощают** задачу, и т.д. Это означает, что лучше вам все же начинать с наземных игр и, всерьез потрудившись над ними и приобретя богатый опыт, переходить к космическим играм.

## Наземные процессоры

Следующим шагом вперед после космических процессоров, с точки зрения визуализации, являются процессоры наземных (ландшафтных) игр. Такой процессор, конечно, предназначен не только для работы с ландшафтом, но ландшафт в таких играх представляет собой важную составляющую игрового мира, и здесь совершенно необходим правильный выбор метода определения видимости многоугольников. На рис. 6.56 приведена копия экрана игры *Tread Marks* фирмы Longbow Digital Arts. Это типичная трехмерная ландшафтная игра. В отличие от ландшафта, объекты в таких играх достаточно просты, и работа с ними не представляет особых трудностей для программиста. Объекты могут перемещаться по поверхности (возможно, летать над ней), и в большинстве случаев это того или иного рода танки или какие-то другие средства передвижения — анимация которых, как правило, является достаточно простой задачей. Конечно, если вы разрабатываете серьезную боевую игру, то вам придется иметь дело со сложной анимацией и отслеживанием перемещения персонажей по неровностям ландшафта. Однако главной трудностью ландшафтных игр тем не менее остается представление базы данных игрового мира, который может быть достаточно большим.

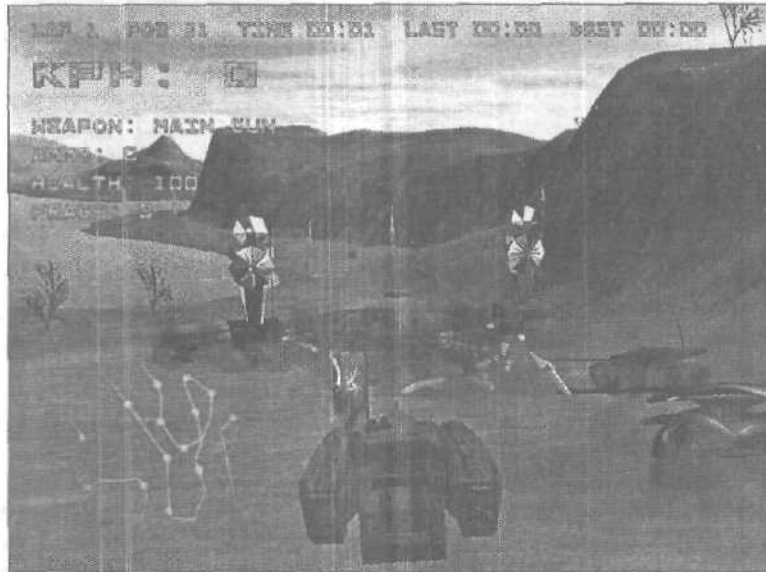


Рис. 6.56. Копия экрана игры *Tread Marks*

Предположим, например, что вы хотите создать игровой мир, который представляет собой сетку размером  $100\,000 \times 100\,000$  единиц, где каждый многоугольник имеет размеры, не превышающие  $200 \times 200$  единиц. Это означает, что у нас имеется сетка многоугольников из порядка

$$\frac{100\,000}{200} \cdot \frac{100\,000}{200} \approx 250\,000 \text{ многоугольников.}$$

Теперь вы понимаете, о чем я хочу сказать? Вы вынуждены отслеживать огромное количество многоугольников, что без той или иной схемы разбиения игрового мира на части практически невозможно. Одна из таких технологий разбиения — секторизация, пример которой показан на рис. 6.57. Здесь я разбил весь игровой мир на матрицу секторов раз-

мером  $8 \times 8$ , в каждом из которых находится порядка 4000 многоугольников ( $250000/64$ ), что представляет собой уже более-менее разумное число. Ну и поскольку в худшем случае — находясь в углу игрового поля — игрок может видеть до 4 секторов, то общее число многоугольников, с которыми придется иметь дело, не превысит порядка 16000, что более-менее терпимо (понятно, что это — без учета объектов игры).

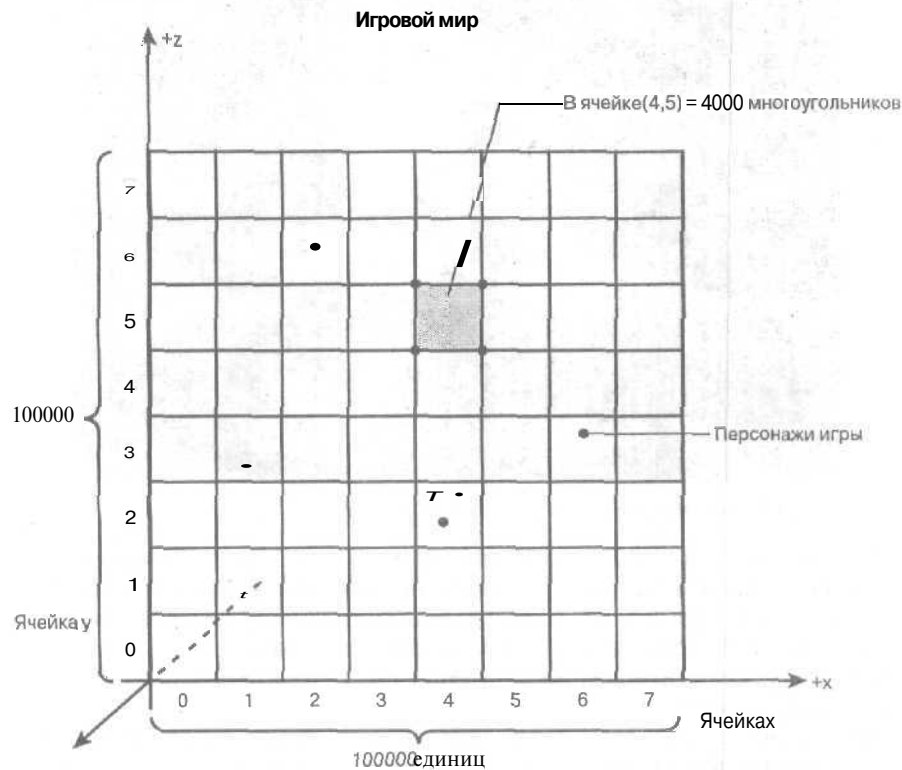


Рис. 6.57. Секторизация большого игрового мира на меньшие блоки

Ландшафтные игры несколько сложнее, чем космические, что связано, как уже говорилось, с большими базами данных ландшафта. Визуализация в такой игре может быть выполнена с помощью модифицированного алгоритма художника или бинарного разбиения пространства (если вы хотите получить действительно большой мир), либо с помощью обычного Z-буфера. Еще одной сложностью данного типа игр является то, что необходимо позаботиться о перемещении объектов по контурам местности, т.е. чтобы ноги (колеса, дно и т.п.) объектов не парили над землей и не проваливались под землю.

## FSP-процессоры

На рис. 6.58 приведена копия экрана игры *Unreal Tournament*, которая представляет собой типичную FSP-игру (first-person shooter, т.е. обычную "стрелялку"). Это наиболее сложный в написании тип игр по целому ряду причин. Во-первых, действие в основном протекает в помещении, что означает высокий уровень детализации. Во-вторых, игровые миры этого типа игр достаточно велики, а соответственно, велики и базы данных многоугольников. Это означает, что просто передать весь мир в трехмерный конвейер визуализации

зации просто невозможно, и игровой мир должен быть **секторизован** для минимизации количества одновременно обрабатываемых многоугольников.



Рис. 6.58. Копия экрана игры Unreal Tournament

Это означает, что нам нужны соответствующие структуры данных и алгоритмы для их обработки. Если в космических или **ландшафтных** играх мы можем пойти на определенную некорректность вида трехмерных моделей, отклонение от сортировки многоугольников и т.п., то в **FPS-играх** это недопустимо. Игрок просто не станет играть в игру, которая выглядит уродливо или некорректно. Игры такого рода просто обязаны выделяться повышенным реализмом.

Соответственно, используемые в такого рода играх физическое моделирование и системы искусственного интеллекта должны быть очень сложными. Одно дело — когда **где-то** далеко в космосе взрывается подбитый космический корабль, который и разглядеть-то толком **невозможно**, и другое — последствия выстрела в упор, когда, например, убитый персонаж падает в воду. Если при этом не будет брызг и **волн**, картина будет выглядеть совершенно нереально.

## Трассировка лучей и воксельные процессоры

О процессорах для работы с многоугольниками было сказано уже немало, но имеются и другие возможные типы игровых процессоров, основанные на трассировке лучей или работе с **вокселями** (так называются по аналогии с пикселями трехмерные элементы **объемного** изображения). **Трассировка лучей** (ray casting) представляет собой технологию, применявшуюся во множестве ранних трехмерных **FPS-игр**, наподобие *Wolfenstein 3D*. Эти игры **использовали прямую трассировку**, т.е. трассировку лучей от точки зрения игрока через плоскость обзора до объекта (рис. 6.59). Такая технология, **использующая** информацию о геометрии игрового мира и объектах, позволяет очень быстро генерировать трехмерные сцены.

НА ЗАМЕТКУ

Я встречался с игрой с системой трехмерной трассировки лучей, написанной **Полем Эдельштейном** (Paul Edelstein) еще в 80-е г.г. прошлого века **для Atari 800**, но он опередил свое время и **последователей** в тот момент у него не нашлось.

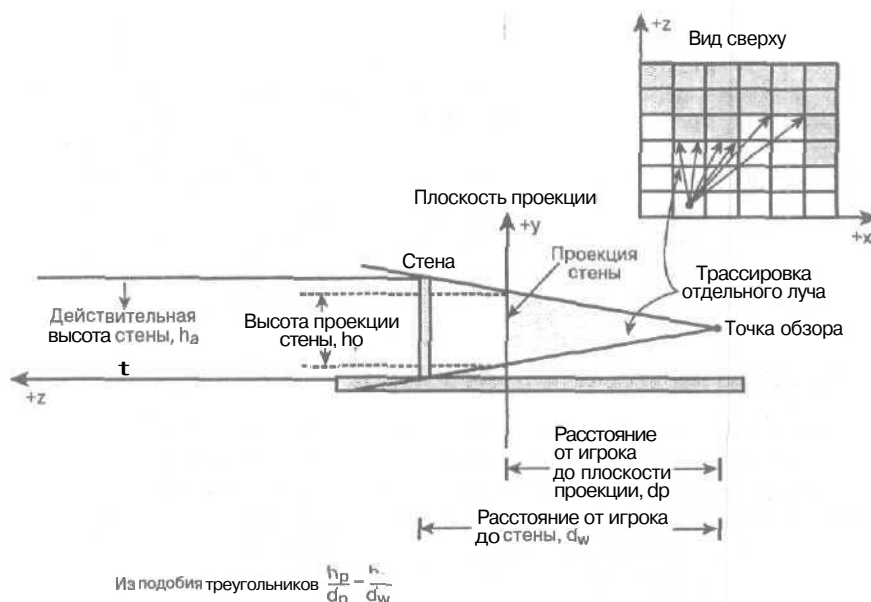


Рис. 6.59. Механизм трассировки лучей

Основная проблема при использовании системы трассировки лучей в ограниченности ее возможностей. Однако я думаю, что мы еще столкнемся в ближайшие годы в играх с новыми современными трассировщиками лучей, которые обеспечат большую степень реализма, чем использование многоугольников.

Еще одна технология отслеживания лучей называется *воксельной графикой*, или *объемной визуализацией*. Одна из наиболее примечательных игр с использованием данной технологии — *Comanche* фирмы Novalogic. Технически воксельная графика использует не трассировку лучей, а методику, называемую *алгоритмом граничащих кубов* (marching cubes algorithm). Для визуализации трехмерного изображения используется множество трехмерных данных, однозначно проецируемых на визуализируемое изображение. Примером использования воксельной системы могут служить различные медицинские системы.

Основная проблема, возникающая при использовании процессоров трассировки лучей и воксельной графики, заключается в том, что обычно требуется очень много данных — действительно, *очень* много. Эти технологии в современных играх практически не используются, поскольку многоугольники с текстурами и эффективнее, и хорошо выглядят. Но, как я уже говорил, мне кажется, что мы вскоре станем свидетелями возврата этих технологий в компьютерные игры.

## Гибридные процессоры

Последний тип процессоров, о которых я хотел рассказать, — это гибридные процессоры. Это процессоры, которые разработаны таким образом, чтобы имитировать и космос, и ландшафт, и даже FPS — все одновременно. Этот термин может также обозначать и процессоры, которые в одной среде объединяют различные технологии. Например, процессор игры *Wolfenstein* использует трассировку лучей для визуализации игрового мира и спрайты для персонажей. *Doom* наряду со спрайтами для персонажей использует многоугольники для представления игрового мира, а персонажи *Quake* представлены как спрайтами, так и многоугольниками (в *Quake III* — только многоугольниками).

Главный вывод, который можно сделать, состоит в том, что при написании игр не следует придерживаться жестких схем, и для решения каждой из подзадач можно использовать свои методы решения, наилучшим образом подходящие для них. В качестве **еще** одного примера можно вспомнить использование в трехмерных играх систем частиц. Зачастую они основаны на использовании многоугольников, которые всегда повернуты к точке обзора наружной поверхностью, т.е. по сути представляют собой трехмерные спрайты.

В заключение, поскольку мы все же **возвращаемся** к процессорам, работающим с многоугольниками, хочу посоветовать вам при желании использовать какие-то иные принципы при написании трехмерных игр обратиться к дополнительной литературе и ознакомиться с другими типами процессоров более подробно.

## Сборка игрового процессора

Мы рассмотрели в этой главе огромное количество самых разнообразных тем, написали немалое количество псевдокода и рассмотрели ряд примеров. Теперь мы готовы к тому, чтобы начать написание рабочей версии кода, которая позволит нам создавать грубые модели трехмерных миров, размещать в них объекты, перемешать их и просматривать при помощи камеры. Мы приступим к этому уже в следующей главе, и в каждой из последующих глав мы будем детально рассматривать те или иные темы. Однако уже в данной главе было написано достаточно кода, чтобы выделить его в отдельный файл `T3DLIB5.CPP`.

В этой главе мы не рассматривали никаких демонстрационных **программ** — только ряд высокоуровневых абстрактных идей с несколькими примерами. Наверное, имеет смысл перед тем, как приступить к чтению очередной главы, еще раз бегло просмотреть данную, чтобы освежить в памяти все представленные в ней идеи.

## Резюме

В этой главе рассмотрено огромное количество материала, но поскольку этого материала действительно очень много, а размер главы ограничен, вероятно, сейчас у вас возникает еще больше вопросов, чем было до **того**, как вы приступили к чтению. Это нормальное явление, и чтение последующих глав, подробно освещающих те или иные аспекты, постепенно даст ответы на все ваши вопросы. Главное в этой главе, на что следует обратить особое внимание, — это использование ряда различных систем координат и преобразования между ними. Далее в книге вы познакомитесь с отдельными темами гораздо более подробно.

# ГЛАВА 7

## Визуализация трехмерных каркасных объектов

### В этой главе...

- Общая архитектура каркасного игрового процессора 516
- Создание простого загрузчика файлов, содержащих трехмерные модели 524
- Разработка трехмерного игрового конвейера 536
- Визуализация трехмерного игрового мира 600
- Трехмерные демонстрационные программы 605

Наконец все готово для создания трехмерных приложений! Вы овладели всеми фундаментальными математическими знаниями и, по крайней мере, самыми общими представлениями о различных частях трехмерного игрового процессора. В данной главе будет разработан несложный игровой *процессор, работающий* в каркасном режиме. В *последующих* главах книги он будет все более усложняться. В него будет добавлена возможность плоского затенения, моделирование *освещения*, отображение *текстуры*, в результате чего мы получим программный игровой процессор, подобный тому, который был разработан для игры *Quake*. Мы будем продвигаться небольшими шагами, чтобы все было понятно. В отличие от предыдущей, данная глава будет изобиловать включениями фрагментов кода; одни и те же действия мы будем пытаться выполнить различными способами. Кроме того, в этой главе мы поработаем над симуляцией камеры и обсудим ее модель, альтернативную модели Эйлера, которая получила название *UVN-системы*. Приведем список вопросов, подлежащих рассмотрению:

- общая архитектура игрового процессора, работающего в каркасном режиме;
- создание простого загрузчика файлов с трехмерными моделями;
- разработка трехмерного игрового конвейера;
- преобразование локальных координат в мировые;
- модели камеры;
- преобразование мировых координат в координаты камеры;
- аксонометрическая проекция координат камеры;
- преобразование аксонометрических координат в экранные;
- удаление лишних объектов;
- удаление обратных поверхностей;
- простое отсечение в двумерном и трехмерном пространствах;
- визуализация трехмерных моделей;

## Общая архитектура каркасного игрового процессора

Перед тем как приступить к непосредственной разработке нашего первого трехмерного игрового процессора, попытаемся четко определить, что же мы хотим получить в результате. Думаю, было бы бессмысленно просто привести игровой процессор *Quake* и сказать: "Вот, любуйтесь". Цель этой книги — шаг за шагом описать процесс последовательной разработки все более сложных версий программного обеспечения, чтобы читатель мог понять каждую тонкость создания трехмерного игрового процессора, начиная с самых основ. Этот процесс подразумевает также, что иногда будут допускаться ошибки и делаться упрощения.

### СОВЕТ

Что касается замечаний об ошибках и упрощениях, следует внести определенные пояснения. Одним из величайших упрощений в науке является модель атома. Из курса средней школы известно, что атом состоит из ядра, в состав которого входят нейтроны и протоны, а также определенного числа электронов, вращающихся вокруг ядра по круговым или эллиптическим орбитам. На самом деле все обстоит намного сложнее. Если быть точным, то не существует каких-либо четко определенных орбит. Можно лишь указать вероятность, с которой электрон находится в той или иной точке пространства. Кроме того, невозможно с абсолютной точностью одновременно зафиксировать положение и импульс электрона. Однако орбитальная модель атома способна объяснить результаты некоторых простых экспериментов. Аналогично, многие методы работы с трехмерными объектами, которые нам предстоит изучить, можно усовершенствовать. Однако если мы не знаем, почему один метод лучше другого, то на этом пути мы обречены повторять подобные ошибки.

В предыдущей главе изложен большой объем теоретического материала, имеющего отношение к трехмерной графике. Вы уже обладаете всеми необходимыми знаниями по трехмерной графике (возможно, за исключением некоторых методов растеризации) и готовы создать неплохой игровой процессор. Теперь мы соберем воедино все математические и алгоритмические концепции, касающиеся систем координат, проекций и структур данных, и разработаем простой игровой трехмерный процессор, работающий в каркасном режиме. В этом процессоре многоугольники можно будет задавать либо как независимые элементы, либо как составляющие объектов более высокого уровня, которые загружаются с диска и подлежат визуализации. Кроме того, мною были разработаны всевозможные функции перехода из одной сис-

темы координат в другую (их на самом деле много), позволяющие **создавать** различные трехмерные игровые конвейеры и экспериментировать с ними. Например, можно загрузить объект, с помощью жестко закодированных преобразований пройти все этапы игрового конвейера и визуализировать объект как набор **многоугольников**. С другой стороны, для **преобразований** при переходах от одного этапа конвейера к другому можно использовать матрицы, разложить объект на **составляющие** его многоугольники, а затем визуализировать их, не **забывая** об исходных объектах. Все это говорится потому, что мы собираемся пройти путь к одной и той же цели не одним, а несколькими путями (системные программисты, работающие в области X Windows, почувствуют себя как **дома!**). Очень скоро вы убедитесь в том, что такие основные трехмерные математические объекты, как матрицы, прекрасны, но абсолютно неприменимы для преобразования из локальных координат в мировые **и т.д.** (по крайней мере, в программных игровых процессорах). Наконец, будут рассмотрены две различные модели камеры. Первая — это обычная эйлерова камера, представляющая собой не что иное, как набор данных, с помощью которых задается ее положение и ориентация. Вторая модель (которая будет описана, но не реализована) — это **UVN-система**, которая задается в терминах, подобных тем, которыми оперирует кинорежиссер. Это направление и фокальная точка, а также ориентация относительно выбранного направления.

Между прочим, если у вас нет современного калькулятора типа Texas Instruments или Casio, купите его. Я даже не могу передать словами, насколько он помог мне в **математических** преобразованиях, используемых в данной книге, и при тестировании. Более того, для таких калькуляторов можно писать ифы, поскольку на многих из них установлены 32-битовые процессоры типа Motorola 68000.

## Структуры данных и трехмерный игровой конвейер

Наш первый игровой трехмерный процессор не будет обладать большими функциональными возможностями. В этой главе последовательно описываются различные стадии его разработки, и в данном разделе для начала грубо очерчивается схема, по которой будут работать источники данных и трехмерный игровой конвейер. Как видно из рис- 7.1, в основе процессора лежат три уникальные структуры данных:

- структура, предназначенная для хранения многоугольников общего вида, основанная на списке вершин;
- структура, предназначенная для автономного многоугольника;
- структура, предназначенная для объекта, состоящего из нескольких многоугольников.

Все они реализованы с помощью структур данных, приведенных в предыдущей главе. Далее они скопированы из предыдущей главы, чтобы освежить вашу память (как и мою — я чуть не свихнулся, подбирая для них понятные имена).

// Многоугольник, основанный на списке вершин

```
typedef struct POLY4DV1_TYP
{
    int state;      // Информация о состоянии
    int attr;       // Физические атрибуты многоугольника
    int color;      // Цвет многоугольника

    POINT4D_PTR vlist; // Список вершин
    int vert[3];       // Индексы списка вершин
} POLY4DV1, *POLY4DV1_PTR;
```

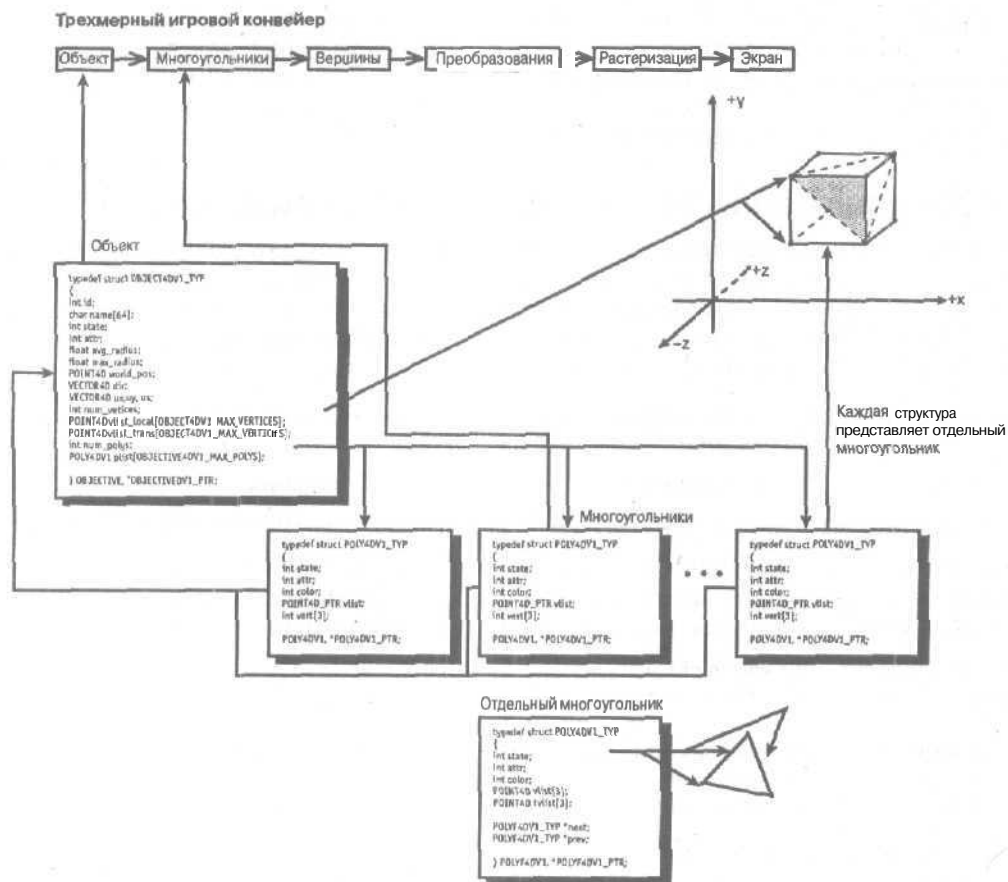
```

// Автономный многоугольник, использующийся
// в списке визуализации
typedef struct POLYF4DV1_TYP
{
    int state;    // Информация о состоянии
    int attr;    // Физические атрибуты многоугольника
    int color;    // Цвет многоугольника

    POINT4D vlist[3];    // Вершины данного треугольника
    POINT4D tvlist[3];    // Вершины после преобразования
                        // (задаются при необходимости)

    POLYF4DV1_TYP *next; // Указатель на следующий
                        // многоугольник в списке
    POLYF4DV1_TYP *prev; // Указатель на предыдущий
                        // многоугольник в списке
} POLYF4DV1, *POLYF4DV1_PTR;

```



ft/c. 7. Л Структуры данных, используемые в трёхмерном игровом процессоре

Приведем также некоторые макроопределения `#define`, облегчающие работу с представленными выше структурами.

```
// Структуры define для многоугольников и поверхностей;  
// версия 1
```

```
// Атрибуты многоугольников и поверхностей  
#define POLY4DV1_ATTR_2SIDED      0x0001  
#define POLY4DV1_ATTR_TRANSPARENT 0x0002  
#define POLY4DV1_ATTR_8BITCOLOR  0x0004  
#define POLY4DV1_ATTR_RGB16      0x0008  
#define POLY4DV1_ATTR_RGB24      0x0010  
  
#define POLY4DV1_ATTR_SHADE_MODE_PURE      0x0020  
#define POLY4DV1_ATTR_SHADE_MODE_FLAT     0x0040  
#define POLY4DV1_ATTR_SHADE_MODE_GOURAUD  0x0080  
#define POLY4DV1_ATTR_SHADE_MODE_PHONG    0x0100  
  
// Состояния многоугольников и поверхностей  
#define POLY4DV1_STATE_ACTIVE      0x0001  
#define POLY4DV1_STATE_CLIPPED    0x0002  
#define POLY4DV1_STATE_BACKFACE  0x0004
```

Некоторые символы могут показаться чуждыми нам, однако они приобретут смысл по мере создания игрового процессора. Далее приведем структуру данных, предназначенную для хранения объекта.

```
// Объект, основанный на списках вершин и многоугольников  
typedef struct OBJECT4DV1_TYP  
{  
    int id;           // Численный идентификатор объекта  
    char name[64];    // ASCII-имя объекта (на всякий случай)  
    int state;        // Состояние объекта  
    int attr;         // Атрибуты объекта  
    float avg_radius; // Средний радиус объекта для  
                     // выявления столкновений  
    float max_radius; // Максимальный радиус объекта  
  
    POINT4D world_pos; // Положение объекта в мировой  
                      // системе координат  
  
    VECTOR4D dir;      // Углы, задающие ориентацию объекта  
                     // в локальных координатах, или  
                     // единичный вектор направления,  
                     // задаваемый пользователем  
  
    VECTOR4D ux,uy,uz; // Локальные оси для отслеживания  
                     // ориентации. При вызове функций  
                     // поворота эти параметры  
                     // автоматически обновляются  
  
    int num_vertices;  // Количество вершин данного объекта  
  
    POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES];
```

```

// Массив локальных вершин
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES];
// Массив вершин после преобразования

int num_polys; // Количество многоугольников в
               // каркасе объекта
POLY4DV1 plist[OBJECT4DV1_MAX_POLYS];
// Массив многоугольников

} OBJECT4DV1, *OBJECT4DV1_PTR;

```

В приведенных выше фрагментах кода нет ничего нового. Есть только две директивы `#define`, которые используются для того, чтобы задавать размер массива. Я установил в них разумные (на мой взгляд) значения.

```

// Чтобы использовать массивы больших размеров,
// измените указанные значения
#define OBJECT4DV1_MAX_VERTICES 64
#define OBJECT4DV1_MAX_POLYS 128

```

НА ЗАМЕТКУ

Было бы эффективнее использовать массивы указателей на многоугольники и вершины, но пока что я не беспокоюсь об экономии памяти. Применение динамических массивов повысило бы сложность алгоритмов.

Продолжая анализировать рис. 7.1, мы видим, что образовался упрощенный трехмерный конвейер, на вход которого подается объект или многоугольник, и который преобразует его из локальных координат через множество промежуточных этапов в экранные координаты. Остановимся на минуту и поговорим о том, что именно будет обрабатываться в трехмерном игровом процессоре.

В большинстве случаев в процессор будет передаваться несколько объектов и (или) отдельных многоугольников, представляющих геометрию игры. На некотором этапе все эти объекты и многоугольники преобразуются в набор многоугольников и помещаются в список визуализации. Однако в более простых трехмерных процессорах есть возможность сохранять все объекты автономными (и целыми), чтобы обрабатывать и визуализировать их как полностью отдельные элементы. В большинстве случаев высокоуровневая иерархическая структура объектов на этапе визуализации не нужна. Функции визуализации трехмерных моделей в основном предпочитают работать с многоугольниками, а не с объектами, однако ничего страшного не произойдет, если они будут реализованы именно таким образом. Возможно, сначала вы придете к выводу, что на этапе растеризации невозможно иметь дело с объектами, поскольку отсутствует единый список многоугольников, в котором представлены все многоугольники, составляющие геометрию игры. Не совсем понятно, как в этом случае производить сортировку многоугольников или заносить их в Z-буфер, чтобы добиться правильного порядка визуализации (конечно же, сам я в этом преуспел).

Суть в следующем. Если сначала сортируются сами объекты, а затем — входящие в их состав многоугольники, то теоретически можно выполнить визуализацию каждого объекта без всяких проблем, потому что каждый объект задает выпуклую область пространства (грубо говоря, в целях сортировки эту область можно представлять в виде ограничивающей сферы или прямоугольного параллелепипеда). Конечно, если функция работает с Z-буфером, то для нее не имеет значения, подается ли в нее список многоугольников или список объектов, состоящих из многоугольников, поскольку на определенном этапе будет производиться растеризация всех элементов, и с помощью Z-буфера будет проверяться порядок вывода на уровне пикселей.

Идея предыдущего абзаца в том, чтобы еще раз подчеркнуть, что одно и то же действие часто будет выполняться несколькими способами. Это позволяет понять преимущества и недостатки различных методов. В результате мне пришлось написать большое количество кода, предназначенного для загрузки и растеризации отдельного многоугольника, или для восприятия набора объектов и сохранения их в целостном виде вплоть до растеризации, или для разбиения объектов на составляющие многоугольники и добавления их в основной список многоугольников, о котором пойдет речь в следующем разделе.

## Основной список многоугольников

В большинстве трехмерных игровых процессоров (программных или аппаратных) все игровые объекты, представленные на высоком уровне (это могут быть трехмерные каркасы, параметрически заданные объекты или объекты, которые генерируются "на лету"), на некотором этапе преобразуются в многоугольники. Эти многоугольники помещаются в основной список или поток, который затем проходит дальнейшую обработку в трехмерном конвейере. Данный список может генерироваться как на этапе, когда все параметры заданы в мировых координатах, так и на последующих этапах конвейера. Возможен и такой вариант, когда все объекты преобразуются в многоугольники после перехода в систему координат камеры. Но рано или поздно наступает момент, когда список многоугольников передается на обработку в функцию растеризации. В наборе программных инструментов нашего трехмерного игрового процессора, конечно же, такой список предусмотрен (рис. 7.2). При желании в него можно поместить все многоугольники. Первая версия структуры данных, которая будет использоваться для хранения списка многоугольников, — это не что иное, как массив указателей на статический массив, что проиллюстрировано на рис. 7.2 и в приведенном ниже определении.

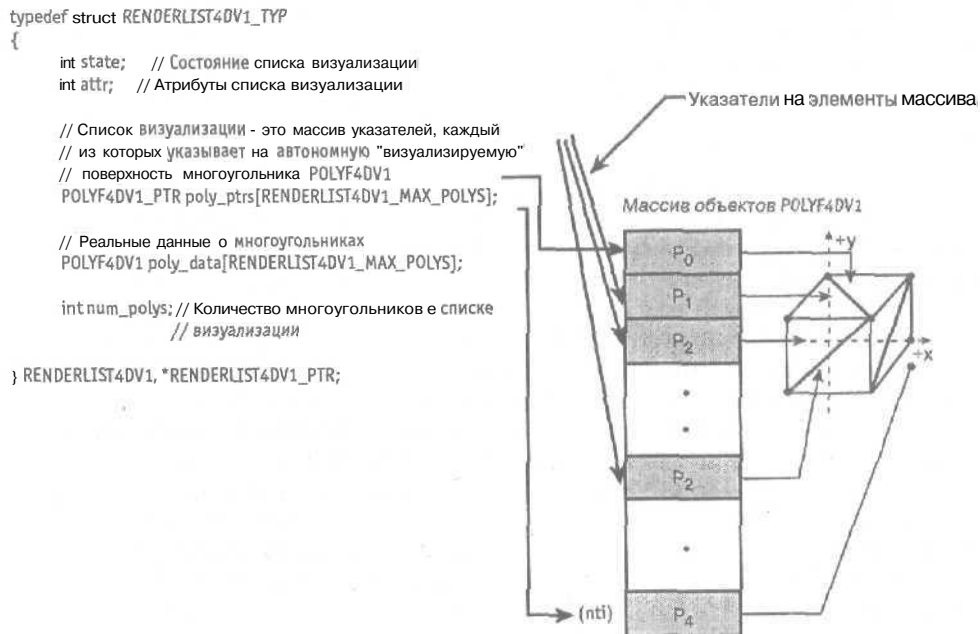


Рис. 7.2. Главный список многоугольников

```

// Объект, предназначенный для хранения списка
// визуализации, - таким образом можно работать сразу с
// несколькими списками
typedef struct RENDERLIST4DV1_TYP
{
    int state;    // Состояние списка визуализации
    int attr;     // Атрибуты списка визуализации

    // Список визуализации - это массив указателей, каждый
    // из которых указывает на автономную визуализируемую
    // поверхность многоугольника POLYF4DV1
    POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

    // Чтобы сэкономить время на выделении и освобождении
    // памяти, предназначенной для многоугольников, из
    // которых состоит каждый каркас, определим хранилище
    // поверхностей многоугольников
    POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

    int num_polys; // Количество многоугольников в списке
                  // визуализации
} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;

```

Как видите, у объекта `RENDERLIST4DV1` есть поля `state` и `attr`, с помощью которых отслеживаются состояние и атрибуты списка (чем мы займемся позже), но действительно важная часть приведенной выше структуры — массивы `poly_ptrs[]` и `poly_data[]`. Они образуют так называемые *косвенный* и *индексированный списки*. Это означает, что сами данные хранятся в массиве `poly_data[]`, но вместо того чтобы непосредственно работать с этим массивом, мы работаем с массивом `poly_ptrs[]`, в котором находятся указатели (он также может состоять из индексов массива `poly_data[]`).

#### НА ЗАМЕТКУ

В технологии `Direct3D` описанный выше метод называется методом с использованием *буферов индексированных вершин* (*indexed vertex buffers*).

Кому-то этот прием может показаться непонятным, однако он необходим по нескольким причинам. Во-первых, если понадобится *отсортировать* многоугольники по определенному признаку, например, по величине координаты `z` (для их визуализации), можно будет ограничиться сортировкой списка `poly_ptrs[]` и не перемешать данные в массиве `poly_data[]`. Во-вторых, чтобы исключить многоугольник из списка `poly_data[]`, пришлось бы удалять все данные и перемешать все остальные элементы, поскольку это был бы единственный имеющийся список. Однако, имея в своем распоряжении массив указателей `poly_ptrs[]`, можно просто удалить связь (и при необходимости сдвинуть все элементы списка вверх), чтобы "выключить" многоугольник в массиве `poly_data[]`. Возможно, кто-то из читателей подумает; "И к чему все эти усложнения? Можно было бы использовать связанный список или *какую-то* другую похожую *конструкцию*". Я с этим согласен, но на данном этапе *проще* иметь дело со статическим массивом, и лишь тогда, когда мы будем знать, что количество многоугольников в списке может меняться от 2 до 2 миллионов, то обратимся к связанному списку. Пока что я не собираюсь обрабатывать в кадре более нескольких сотен многоугольников, поэтому могу статически выделить для них память и ни о чем не беспокоиться.

Наконец, поскольку возможно наличие нескольких списков *визуализации*, соответствующих различным камерам, было решено удержаться от применения в списке *визуали-*

зации глобальных переменных. Поэтому, чтобы определить список визуализации, нужно воспользоваться инструкцией наподобие *следующей*.

```
RENDERLIST4DV1 rend_list;
```

После этого следует вызвать функцию, выполняющую сброс этого списка в процессе подготовки к формированию следующего кадра анимации.

```
void Reset_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list)
```

```
{  
    // Эта функция инициализирует и обнуляет элементы  
    // переданного б нее списка и готовит их для размещения  
    // параметров многоугольников и поверхностей. Заметим,  
    // что в данной версии список визуализации состоит из  
    // указателей на объекты FACE4DV1, поэтому она  
    // вызывается для каждого кадра  
  
    // Поскольку количество многоугольников в списке  
    // отслеживается с помощью переменной num_polys, ей  
    // можно присвоить значение 0. Позже, когда мы откажемся  
    // от списка указателей и перейдем к использованию  
    // связанного списка, нам понадобится более надежная  
    // схема  
    rend_list->num_polys = 0;  
  
} // Reset_RENDERLIST4DV1
```

Позже, когда будет создан более динамичный список визуализации (если он нам понадобится), эту функцию, конечно же, придется усложнить, поскольку она, *вероятно*, будет должна освобождать память и обнулять данные, оставшиеся от многоугольников предыдущего кадра.

НА ЗАМЕТКУ

Структуры данных крайне важны в трехмерной графике. Невозможно *переоценить*, как важно уметь обращаться со связанными списками, *двусвязными* списками, деревьями, графами, а также различными алгоритмами поиска, удаления и другими операциями. В книге *Программирование игр для Windows. Советы профессионала* описано несколько таких структур данных, однако существует более полная литература, посвященная этим вопросам. Среди книг, посвященных данным темам, можно упомянуть книги Роберта Седжвика (Robert Sedgewick) *Algorithms in C++* и ЛиндERTA Аммераала (Leendert Ammeraal) *Programs and Data Structures in C*.

Теперь, когда мы ознакомились со структурами данных, все готово для того, чтобы приступить к разработке трехмерного каркасного процессора. Забавно, что на самом деле осталось не так много работы (конечно же, нам придется иметь дело со множеством технических деталей и связей между частями программы, однако нам не потребуется прилагать такие невероятные умственные усилия, как при заполнении налоговой декларации). Все, что нужно, — это создать многоугольник или загрузить объект (для этого понадобится специальная функция), а затем — пройти такие этапы.

1. Перейти из локальных координат в мировые.
2. Удалить лишние объекты (необязательный этап).
3. Удалить обратные поверхности (необязательный этап).
4. Перейти из мировых координат в координаты камеры.
5. Выполнить отсечение в трехмерном пространстве (необязательный этап).

6. Выполнить аксонометрическое преобразование, чтобы учесть перспективу.
7. Перейти из аксонометрических координат в экранные.
8. Растеризовать многоугольники.

НА ЗАМЕТКУ

Иногда этапы 6 и 7 объединяют в один.

Конечно же, некоторые этапы можно поменять местами. Кроме того, имеет немалое значение, на каких этапах мы имеем дело с объектами, а на каких они уже преобразованы в списки многоугольников (на определенном этапе работы конвейера это преобразование нужно будет выполнить обязательно). Наконец, есть различные способы перехода из одной системы координат в другую: с помощью матриц, вручную и смешанный способ. Например, если преобразование локальных координат в мировые выполняется с помощью матриц, то возрастает количество ненужных **операций** умножения, поскольку то же самое можно сделать, применив буквально три операции сложения, **приходящихся** на одну точку. В то же время матричное умножение потребует от нас 16 умножений и 12 сложений. С другой стороны, преобразование в систему координат камеры почти всегда производится с **помощью** матриц, потому что в данном случае они не являются разреженными. Таким образом, можно сделать вывод, что не существует универсального способа, по которому можно было бы сделать все необходимые действия. Если выбранный метод работает, причем работает **быстро**, и если он вам подходит — значит, он правильный! Однако в данной главе очень часто одна и та же цель будет достигаться несколькими способами. Это позволит вам прочувствовать работу различных методов применительно к игровому конвейеру и понять, насколько они могут оказаться непригодными в некоторых ситуациях. Особенно это касается однородных координат, которые, в зависимости от обстоятельств, могут быть как удивительно хорошими, так и не менее плохими!

## Новые программные модули

Все новые функции, которые используются в данной **главе**, содержатся в библиотечных модулях:

- **T3DLIB5.H** — заголовочный файл;
- **T3DLIB5.CPP** — исходный файл на C/C++.

Вряд ли стоит упоминать о том, что для компиляции любой программы, иллюстрирующей материал настоящей главы, понадобится подключить все **предыдущие** библиотечные файлы **T3DLIB\*.CPP|H**, а также библиотеки DirectX, и настроить компилятор так, чтобы он создавал исполняемые файлы Win32. Кроме того, я всегда включаю в листинг каждой демонстрационной программы одну-две строки комментария, в котором перечисляются **файлы**, необходимые для компиляции данной программы. Поэтому, если возникают сомнения по поводу необходимости подключения того или иного библиотечного файла, всегда можно обратиться к комментариям.

## Создание простого загрузчика файлов с трехмерными моделями

В предыдущей главе описано несколько форматов, предназначенных для хранения трехмерных моделей. По-видимому, вы уже экспериментировали с одной из трехмерных моделей, содержащихся на прилагаемом к книге компакт-диске, или одной из тех, кото-

рую вы сами нашли. Можно попробовать экспортировать простой объект, например, куб, и ознакомиться с получившимися данными в ASCII-редакторе. Написание загрузчика файлов с трехмерными моделями — достаточно простая задача для тех, кто знаком с устройством компилятора и синтаксического анализатора. Для остальных разработка синтаксического анализатора может послужить ценным опытом, потому что это не та область, в которой что-то можно делать наспех. Если вам приходится интенсивно работать с трехмерными моделями, то может возникнуть желание найти где-нибудь набор готовых функций, выполняющих импорт/экспорт и преобразование моделей в различных форматах, чтобы не пришлось писать их самому. Однако я люблю все делать сам, поэтому собираюсь создать загрузчики файлов для всех форматов, которые используются в этой книге, а не брать готовые. Я попытаюсь, чтобы они получились короткими, понятными и соответствующими поставленной задаче.

#### ВНИМАНИЕ

Только сегодня я уже получил около 13 электронных сообщений с жалобами на то, что при работе компоновщика возникает ошибка, связанная с невозможностью найти функцию `main()`. Если я специально не указываю, что по определенным причинам следует создать консольное приложение, это означает, что следует создавать приложения Win32, в роли входной точки в которых выступает функция `WinMain()`. Единственная возможная причина, по которой в компоновщике возникает упомянутая ошибка, — это компоновка приложения Windows как КОНСОЛЬНОГО приложения, когда компилятор настроен для создания КОНСОЛЬНЫХ приложений. Я собираюсь организовать группу под названием “ГЛНИК” — Группа Любителей Неправильного Использования Компилятора. Если вы заинтересованы в том, чтобы стать членом этой группы, — присылайте заявки по электронному адресу [compUerabuser@xgames3d.com](mailto:compUerabuser@xgames3d.com).

Учитывая сказанное, начнем с форматов .PLG и .NFF, поскольку разобраться с ними проще всего, а это важный аргумент. Кроме того, эти форматы удобочитаемы, файлы с ними можно создать вручную — чем мы и займемся в данной главе. Зачем это нам нужно? Просто я хочу, чтобы вы сами научились работать с трехмерными координатами перед тем, как порекомендовать эту работу программе, предназначенной для создания трехмерных моделей.

Приступим к разработке загрузчика файлов в формате .PLG. Напомню, что в основе данного формата лежит список вершин, а список многоугольников содержит индексы списка вершин. Приведем небольшой файл в формате .PLG, чтобы вы смогли получить о нем общее представление.

```
# Это комментарий
# Строка заголовка: имя объекта, количество вершин
#               и многоугольников
имя_объекта количество_вершин количество_многоугольников

# Список вершин
# Все вершины задаются в виде x y z
x0 y0 z0
x1 y1 z1
x2 y2 z2
x3 y3 z3
*
*
xn yn zn

tf Список многоугольников
# Все многоугольники задаются в таком виде:
```

```

дескриптор_поверхности количество_вершин vO v1 . vn
дескриптор_поверхности количество_вершин vO v1 . vn
дескриптор_поверхности количество_вершин vO v1 . vn
дескриптор_поверхности количество_вершин vO v1 . vn
.
дескриптор_поверхности количество_вершин vO v1 . vn

# Здесь vO v1 . vn - количество вершин, из которых состоит
#каждый многоугольник

А вот каким образом выглядит простой кубический объект.

# Начало файла в формате plg/plx

# Обычный куб, 8 вершин, 12 многоугольников
t1 8 12

# Список вершин в формате координат x y z
5 5 5
-5 5 5
-5 5 -5
5 5 -5
5 -5 5
-5 -5 5
-5 -5 -5
5 -5 -5

# Список многоугольников
# В каждом многоугольнике - по 3 вершины
0xd0f0 3 0 1 2
0xd0f0 3 0 2 3
0xd0f0 3 0 7 4
0xd0f0 3 0 3 7
0xd0f0 3 4 7 6
0xd0f0 3 4 6 5
0xd0f0 3 1 6 2
0xd0f0 3 1 5 6
0xd0f0 3 3 6 7
0xd0f0 3 3 2 6
0xd0f0 3 0 4 5
0xd0f0 3 0 5 1

```

Единственный недостаток формата **PLG** — это то, что он несколько устарел. Поэтому я немного изменил его, добавив дополнительные функциональные возможности, и назвал новый формат **PLX**. Если взять программу, создающую трехмерные модели, и экспортировать полученные с помощью ее модели в формат **.PLG**, наш загрузчик будет в состоянии читать эти файлы, однако 16-битовый дескриптор поверхностей окажется некорректным. Ниже приведено определение нового дескриптора поверхности, используемого в модифицированных файлах в формате **.PLG** (который я буду называть форматом **.PLX**).

Количество битов:  $d_{15}$   $d_0$   
Кодирование: CSSD | RRRR | GGGG | BBBB

где

- C — флаг цвета в формате RGB или в виде индекса;
- SS — два бита, с помощью которых задается режим затенения;
- D — флаг, определяющий двусторонние поверхности;
- RRRR, GGGG, BBBB — биты, с помощью которых задаются интенсивности красного, зеленого и синего каналов;
- GGGGBBBB — байт, интерпретируемый как индекс цвета в 8-битовом режиме.

Приведем макроопределения `#define`, с помощью которых можно различать и извлекать эти битовые поля.

// Битовые маски, упрощающие тестирование

```
// Маска извлечения цвета в формате RGB или в виде индекса
#define PLX_RGB_MASK    0x8000
// Маска извлечения режима затенения
#define PLX_SHADE_MODE_MASK 0x6000
// Маска задания двусторонности
#define PLX_2SIDED_MASK  0x1000
#define PLX_COLOR_MASK   0x0fff // xxxrrrrggggbbbb,
                                // по 4 бита на каждый канал RGB
                                // xxxxxxxxiiiiii - 8-битовый
                                // индекс в индексном режиме
```

// Флаги сравнения, используемые после наложения маски

```
// Цветовой режим многоугольника
// Цвет многоугольника задается в формате RGB
#define PLX_COLOR_MODE_RGB_FLAG  0x8000
// 8-битовый индексный цвет многоугольника
#define PLX_COLOR_MODE_INDEXED_FLAG 0x0000
```

```
// Флаги двусторонности
// Многоугольник двусторонний
#define PLX_2SIDED_FLAG    0x1000
// Многоугольник односторонний
#define PLX_1SIDED_FLAG    0x0000
```

```
// Режим затенения многоугольника
// Постоянное затенение
#define PLX_SHADE_MODE_PURE_FLAG  0x0000
// Плоское затенение
#define PLX_SHADE_MODE_FLAT_FLAG  0x2000
// Затенение по Гуро
#define PLX_SHADE_MODE_GOURAUD_FLAG 0x4000
// Затенение по Фонгу
#define PLX_SHADE_MODE_PHONG_FLAG  0x6000
```

Например, чтобы задать многоугольник с двусторонней поверхностью в 16-битовом режиме, цветом (0x3,0x8,0xF) в формате RGB и затенением по Гуро, нужно сконструировать дескриптор поверхности, приведенный ниже.

Дескриптор 16-битовой поверхности в формате CSSD|RRRR|GGGG|BBBB, где C = 1, что соответствует 16-битовому цветовому режиму, SS = 10, что соответствует затенению по Гуро, D = 1, что соответствует двусторонней поверхности.

Таким образом, полубайт CSSD выглядит как

1101

Значения полубайтов RRRR|GGGG|BBBB равны 0x3, 0x8 и 0xf. В результате получаем следующий дескриптор:

1101 0011 1000 1111 = 0xD38F

Вот и все. В четырех старших битах каждого дескриптора описываются свойства многоугольника и его цветовая модель, а в остальных 12 битах — цвет. Все, что нужно сделать загрузчику файлов, это разделить биты и присвоить надлежащие значения полям структур данных, в которые загружается объект. Рассмотрим функцию, предназначенную для считывания файлов в формате .PLG.

## Загрузчик файлов в формате .PLG/X

Приятная особенность файлов в формате .PLG заключается в том, что их формат фиксирован; это означает, что в них нельзя сначала помещать многоугольники, а затем — вершины. Сначала задаются параметры вершин, и только после этого — параметры многоугольников (файлы в формате .DXF, например, имеют более свободную форму). Каждая строка данных .PLG-файла должна быть определена в корректном формате, обычно представляющем собой одно или несколько чисел, разделенных пробелами. Поэтому план заключается в том, чтобы написать простой синтаксический анализатор, поочередно считывающий каждую строку, а также незамысловатую систему с конечным числом состояний, которой известно, что файл в формате .PLG состоит из трех разделов:

раздел 1 — заголовочная информация;

раздел 2 — параметры вершин;

раздел 3 — параметры многоугольников.

Каждый раздел обладает своим собственным форматом и представляет собой набор данных в виде строк. При этом гарантируется, что каждая порция данных не переносится на следующую строку, так что она не может занимать несколько строк. Таким образом, можно написать синтаксический анализатор, который считывает строки по одной, а затем, в зависимости от своего состояния (т.е. с каким разделом он работает), анализирует строки и извлекает из них данные (разделенные пробелами). Вот и все, что можно сказать по этому поводу. По мере того, как синтаксический анализатор производит разбор данных, результаты помещаются в определенные структуры данных (в данном случае это объект OBJECT4DV1).

Синтаксический анализатор можно было бы реализовать в виде одной функции, но такая функция получилась бы слишком запутанной, учитывая множество различных ситуаций, с которыми ей бы пришлось иметь дело. Поэтому я решил разделить эту функцию на две.

### СОВЕТ

Правильный подход к синтаксическому анализу мог бы заключаться в разработке набора соответствующих функций, которые можно было бы скомпоновать в программу, считывающую файлы с учетом спецификаций языка. Конечно же, существуют различные программные инструменты, предназначенные для разработки компиляторов и интерпретаторов, включая такие как YACC и LEX.

Первая функция поддержки, `Get_Line_PLG()`, поочередно считывает по одной строке файла в формате `.PLG` (ее работа основана на использовании функции `fgets()`). Код этой функции приведен ниже.

```
char *Get_Line_PLG(char *buffer, int maxlength, FILE *fp)
{
    // Эта небольшая вспомогательная функция считывает
    // поочередно по одной строке из файла в формате PLG и
    // пропускает комментарии и пустые строки. Функция
    // возвращает значащие строки обрабатываемого файла или
    // значение NULL, если этот файл пуст

    int index - 0; // Общий индекс
    int Length - 0; // Общая длина

    // Входим в цикл синтаксического анализа
    while(1)
    {
        // Считываем очередную строку
        if (!fgets(buffer, maxLength, fp))
            return(NULL);

        // Удаляем пробел
        for(length = strlen(buffer), index = 0;
            isspace(buffer[index]); index++);

        // Проверяем, не является ли эта строка пустой
        // строкой или комментарием
        if (index >= length || buffer[index] == '#')
            continue;

        // Теперь понятно, что считанная строка является
        // значащей
        return(&buffer[index]);
    } // while
} // Get_Line_PLG
```

Как видим, в этой функции нет ничего особенного. Она умеет различать строки, в которых находятся одни пробелы, и строки комментариев, определяемые символом `#`. Функция просто возвращает следующую строку файла или значение `NULL`, если файл пустой. Последний вариант возникает лишь в случае сбоя, поскольку в файле в формате `.PLG` должно содержаться заданное число полубайтов данных, соответствующих количеству, указанному в заголовочной информационной строке. Функция, которая считывает файл в формате `.PLG`, получила имя `Load_OBJECT4DV1_PLG()` (обратите внимание на соглашение о присвоении имен). Ниже приведен код этой функции.

```
int Load_OBJECT4DV1_PLG(
    OBJECT4DV1_PTR obj, // Указатель на объект
    char *filename, // Имя файла в формате plg
    VECTOR4D_PTR scale, // Начальный масштаб
    VECTOR4D_PTR pos, // Начальное положение
    VECTOR4D_PTR rot) // Начальная ориентация
{
```

```

// Функция загружает с диска объекты в формате .PLG, а
// также позволяет задавать в вызывающей программе
// масштаб, положение и ориентацию объекта, чтобы
// избежать лишних вызовов для статических объектов

FILE *fp; // Указатель на файл
char buffer[256]; // Рабочий буфер

char *token_string; // Указатель на лексему; подготовка
// к анализу

// Краткое описание формата. Обратите внимание на тип в
// конце каждого описания
// # Комментарий.

// # Дескриптор объекта.
// имя_объекта_string кол._вершин_int
// кол._многоугольников_int

// # Список вершин.
// x0_float y0_float z0_float
// x1_float y1_float z1_float
// x2_float y2_float z2_float
// .
// .
// xn_float yn_float zn_float
//
// # Список многоугольников
// описание_поверхности_ushort кол._вершин_int
// v0_index_int
// v1_index_int
// -
// vn_index_int
// .
// .
// Описание_поверхности_ushort кол_вершин_int
// v0_index_int
// v1_index_int
// .
// vn_index_int
// .
// .

// Для простоты предположим, что в каждой строке
// содержится по одному элементу. Таким образом, нужно
// найти дескриптор объекта, считать его, затем найти и
// считать список вершин и, наконец, список
// многоугольников

// Этап 1: обнуление и небольшая инициализация объекта
memset(obj, 0, sizeof(OBJECT4DV1));

// Задаем состояние объекта как активное и видимое

```

```

obj->state = OBJECT4DV1_STATE_ACTIVE |
            OBJECT4DV1_STATE_VISIBLE;

// Задаем положение объекта
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

// Этап 2: открываем файл для чтения
if (!(fp = fopen(filename, "r")))
{
    printf("Couldn't open PLG file %s.", filename);
    return(0);
} // if

// Этап 3: извлекаем первую лексему, которая должна быть
// дескриптором объекта
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
{
    Write_Error("PLG file error with file %s "
               "(object descriptor invalid).",
               filename);
    return (0);
} // if

Write_Error("Object Descriptor: %s", token_string);

// Анализируем параметры объекта
sscanf(token_string, "%s %d %d", obj->name,
        &obj->num_vertices, &obj->num_polys);

// Этап 4: загружаем список вершин
for (int vertex=0; vertex<obj->num_vertices; vertex++)
{
    // Извлекаем очередную вершину
    if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
    {
        Write_Error("PLG file error with file %s "
                   "(vertex list invalid).", filename);
        return(0);
    } // if

    // Анализируем вершину
    sscanf(token_string, "%f %f %f",
           &obj->vlist_local[vertex].x,
           &obj->vlist_local[vertex].y,
           &obj->vlist_local[vertex].z);
    obj->vlist_local[vertex].w = 1;

    // Выполняем масштабирование вершин
    obj->vlist_local[vertex].x*=scale->x;
    obj->vlist_local[vertex].y*=scale->y;

```

```

obj->vlist_local[vertex].z*=scale->z;

Write_Error("\nVertex %d - %f, %f, %f", vertex,
obj->vlist_local[vertex].x,
obj->vlist_local[vertex].y,
obj->vlist_local[vertex].z,
obj->vlist_local[vertex].w);

} // for vertex

// Вычисляем средний и максимальный радиусы
Compute _OBJECT4DV1_Radius(obj);

Write_Error("\nObject average radius - %f, "
"max radius = %f",
obj->avg_radius, obj->max_radius);

int poly_surface_desc - 0; // Дескриптор поверхности
// в формате PLG/PLX
int poly_num_verts = 0; // Количество вершин в
// текущем многоугольнике (их
// всегда 3)
char tmp_string[8]; // Временная переменная для
// хранения дескриптора
// поверхности; проверяем,
// нужно ли преобразовать его
// из шестнадцатеричной
// системы счисления

// Этап B: загружаем список многоугольников
for (int poly=0; poly < obj->num_polys; poly++)
{
// Извлекаем дескриптор очередного многоугольника
if (!(token_string - Get_Line_PLG(buffer, 255, fp)))
{
Write_Error("PLG file error with file %s "
"(polygon descriptor invalid).",
fi lename);
return (0);
} // if

Write_Error("\nPolygon %d:", poly);

// В каждом списке вершин ДОЛЖНО содержаться 3
// вершины, поскольку мы придерживаемся правила,
// согласно которому все модели должны состоять из
// треугольников; считываем дескриптор поверхности,
// количество вершин и их список
sscanf(token_string, "%s %d %d %d %d", tmp_string,
&poly_num_verts, // Всегда равно 3.
&obj->plist[poly].vert[0],
&obj->plist[poly].vert[1],
&obj->plist[poly].vert[2]);

```

```

// Поскольку дескриптор поверхности может быть в
// шестнадцатеричном формате (при этом в его начале
// стоят символы "0x"), нужно провести проверку
if (tmp_string[0] — '0' &&
    toupper(tmp_string[1]) — 'X')
    sscanf(tmp_string, "%x", &poly_surface_desc);
else
    poly_surface_desc = atoi(tmp_string);

// Указываем из списка вершин многоугольника на
// список вершин объекта. Заметим, что это излишне,
// поскольку список многоугольников содержится в
// объекте, и пользователь волен выбрать, какой
// список применяется при построении многоугольника
// - локальный или преобразованный. Для
// многоугольников, входящих в состав объекта, для
// этого параметра лучше задать значение NULL
obj->plist[poly].vlist = obj->vlist_local;

Write_Error("\nSurface Desc - 0x%.4x, num_verts "
    "= %d, vert_indices [%d, %d, %d]",
    poly_surface_desc,
    poly_num_verts,
    obj->plist[poly].vert[0],
    obj->plist[poly].vert[1],
    obj->plist[poly].vert[2]);

// Теперь, когда загружен список вершин, а также их
// индексы, проанализируем дескриптор поверхности и
// на его основе зададим поля многоугольника

// Извлекаем из дескриптора поверхности все поля;
// сначала решим вопросы, связанные с
// односторонностью или двусторонностью поверхностей
if ((poly_surface_desc & PLX_2SIDED_FLAG))
{
    SET_BIT(obj->plist[poly].attr,
        POLY4DV1_ATTR_2SIDED);
    Write_Error("\n2 sided.");
} // if
else
{
    // Односторонняя поверхность
    Write_Error("\n1 sided.");
} // else

// Зададим цветовой режим и значение цвета
if ((poly_surface_desc & PLX_COLOR_MODE_RGB_FLAG))
{
    // Поверхность в режиме RGB 4.4.4
    SET_BIT(obj->plist[poly].attr,

```

```
POLY4DV1_ATTR_RGB16);
```

```
// Извлекаем цвет и копируем его в переменную,  
// хранящую цвет многоугольника в надлежащем  
// 16-битовом формате. Это формат 0x0RGB, в  
// котором на каждый пиксель отводится по 4 бита  
int red - ((poly_surface_desc & 0x0f00) >> 8);  
int green - ((poly_surface_desc & 0x00f0) >> 4);  
int blue - (poly_surface_desc & 0x000f);
```

```
// Данные представляются в формате 4.4.4, а  
// графическая карта работает либо в формате  
// 5.5.5, либо в формате 5.6.5. Наша виртуальная  
// система формирования цвета преобразует формат  
// 8.8.8 в формат 5.5.5 или в формат 5.6.5.  
// Однако перед этим нужно преобразовать все  
// значения, заданные в формате 4.4.4, в формат  
// 8.8.8
```

```
obj->plist[poly].color =  
    RGB16Bit(red*16, green*16, blue*16);  
Write_Error("\nRGB color - [%d, %d, %d]",  
    red, green, blue);
```

```
} // if
```

```
else
```

```
!
```

```
// Цвет поверхности задан в 8-битовом режиме  
SET_BIT(obj->plist[poly].attr,  
    POLY4DV1_ATTR_8BITCOLOR);
```

```
// Просто извлекаем последние 8 битов; это и  
// будет индекс цвета  
obj->plist[poly].color =  
    (poly_surface_desc & 0x00ff);
```

```
Write_Error("\n8-bit color index - %d",  
    obj->plist[poly].color);
```

```
} // else
```

```
// Обрабатываем режим затенения  
int shade_mode - (poly_surface_desc &  
    PLX_SHADE_MODE_MASK);
```

```
// Задаем режим затенения многоугольника  
switch(shade_mode)
```

```
{  
    case PLX_SHADE_MODE_PURE_FLAG: {  
        SET_BIT(obj->plist[poly].attr,  
            POLY4DV1_ATTR_SHADE_MODE_PURE);  
        Write_Error("\nShade mode - pure");  
    } break;
```

```
    case PLX_SHADE_MODE_FLAT_FLAG: {
```

```

        SET_BIT(obj->plist[poly].attr,
            POLY4DV1_ATTR_SHADE_MODE_FLAT);
        Write_Error("\nShade mode=flat");
    } break;

    case PLX_SHADE_MODE_GOURAUD_FLAG: {
        SET_BIT(obj->plist[poly].attr,
            POLY4DV1_ATTR_SHADE_MODE_GOURAUD);
        Write_Error("\nShade mode - gouraud");
    } break;

    case PLX_SHADE_MODE_PHONG_FLAG: {
        SET_BIT(obj->plist[poly].attr,
            POLY4DV1_ATTR_SHADE_MODE_PHONG);
        Write_Error("\nShade mode = phong");
    } break;

    default: break;
} // switch

// Задаем активное состояние многоугольника
obj->plist[poly].state = POLY4DV1_STATE_ACTIVE;

} // for poly

// Закрываем файл
fclose(fp);

// Код успешного выполнения
return(1);

} // Load_OBJECT4DV1_PLG

```

Функция выполняет определенную проверку на наличие ошибок. Возможно, с этим я немного перестарался, но мне просто хотелось сразу привить вам привычку добавлять подобную проверку во все функции, предназначенные для загрузки файлов, потому что иначе все может пойти *наперекосяк* — а ведь в функции производится запись данных в память. Как правило, я никогда не выполняю обработку ошибок в коде самой игры, поскольку, если мой код несовершенен, то я просто довожу его до совершенства. Другое дело — загрузчики файлов... Вы рассчитываете на то, что внешний файл корректен, но кто сказал, что это так и есть? Ведь такой файл может быть получен из самых разных источников. Сообщения об обнаруженных в процессе загрузки данных ошибках заносятся в специальный файл.

Рассмотрим принцип работы приведенной выше функции. Сначала она открывает файл в формате *.PLG*, указанный в строке *filename*, затем считывается первая строка, в которой указано количество вершин и многоугольников, которые предстоит считать. Далее функция переходит к последовательному считыванию списка вершин, помещая их в переданный в функцию объект *OBJECT4DV1*. После этого она переходит к многоугольникам и заносит их в память вместе с их флагами и цветовыми режимами, которые задаются с помощью дескриптора поверхности каждого многоугольника. Дескрипторы преобразуются во флаги вида *POLY\_ATTR\_\** и объединяются с помощью побитового оператора *ИЛИ*. Кроме того, в функцию можно передать дополнительные параметры, позволяю-

щие преобразовать загружаемый объект. Это  $scale$ ,  $pos$  и  $rot$ , представляющие собой начальный масштаб, положение и ориентацию объекта (заметьте, что эти величины являются векторами).

Еще одна интересная особенность данной функции — она вычисляет средний и максимальный радиусы объекта, которые на последующих этапах игрового конвейера будут использоваться для выявления столкновений объекта и его удаления. Чтобы загрузить объект с диска, нужен примерно следующий код.

```
OBJECT4DV1 obj;           // Хранилище объекта.

VECTOR4D scale = {1,1,1,1}, // Масштабирование объекта не
                           // выполняется
pos = {0,0,0,1},           // Положение в мировых
                           // координатах (0,0,0)
rot = {0,0,0,1};           // Объект не поворачивается

// Загрузка объекта
Load_OBJECT4DV1_PLG(&obj, "cube.plg", &scale, &pos, &rot);
```

Вот и все. Воспользовавшись отладчиком или другим программным инструментом, позволяющим заглянуть в структуру данных `obj`, можно увидеть, что в нее считаны параметры объекта, причем все сделано правильно (я надеюсь!). В результате загрузки внешнего файла в объект класса `OBJECT4DV1` инициализируется сам объект, а его полям `state` и `attr` присваиваются корректные значения. Кроме того, задается начальный базис ориентации, определяемый переменными `ix`, `iy`, `iz` (в нашем случае  $ix = \langle 1, 0, 0, 1 \rangle$ ,  $iy = \langle 0, 1, 0, 1 \rangle$  и  $iz = \langle 0, 0, 1, 1 \rangle$ , что соответствует выравниванию по осям мировой системы координат). Теперь давайте сориентируемся, как далеко мы продвинулись в плане обработки геометрии. На данном этапе созданы три различные структуры данных, которые можно применять в дальнейшем;

- `POLY4DV1` — отдельный многоугольник, для которого нужен внешний список вершин;
- `POLYF4DV1` — отдельный многоугольник, обладающий своим собственным хранилищем вершин; он служит базой для основного списка многоугольников;
- `OBJECT4DV1` — отдельный объект, состоящий из некоторого количества многоугольников; в нем содержится собственное хранилище для вершин и многоугольников, поэтому он самодостаточен.

Можно было бы создать игровой процессор, способный обрабатывать все перечисленные типы на всех этапах игрового конвейера. Однако было решено, что процессор будет работать только со структурами `POLYF4DV1` и `OBJECT4DV1`, поскольку `POLY4DV1` — это то же самое, что и `POLYF4DV1`, только без списка вершин. Итак, приступим к разработке процессора, заставляющего эти многоугольники пройти по всем этапам игрового конвейера!

## Разработка трехмерного игрового конвейера

Итак, мы готовы приступить к реализации трехмерного игрового конвейера. В нашем распоряжении имеются структуры данных, и мы научились загружать внешние объекты в формате `.PLG` (фактически в усовершенствованном формате `.PLG`). Теперь всего-навсего осталось написать эти функции (каких-то несколько тысяч строк — было бы о чем говорить!).

Единственное, о чем я *сожалею*, — прежде чем мы доберемся до демонстрационных программ, способных выводить что-нибудь на экран, придется написать большое количество функций, потому что все они связаны между собой. Однако на данном этапе я поступлю следующим образом. Я создам несколько демонстрационных программ, которые будут иллюстрировать некоторые подготовительные методы, предшествующие стадии визуализации.

## Функции преобразования общего вида

Напомним, что все действия в трехмерном пространстве можно выполнять с помощью матриц преобразования  $4 \times 4$  с последующим преобразованием четырехмерных однородных координат в трехмерные (путем деления на  $w$ ). Однако это еще не означает, что это самый быстрый путь! В **общем** случае любое преобразование, представляющее собой совокупность поворотов, перемещений и проекций, лучше выполнять с помощью единой матрицы  $4 \times 4$ , поскольку она представляет сразу несколько собранных воедино преобразований и **обычно** не является разреженной. Если же вы просто пытаетесь сместить в пространстве точку  $p$ , делать это с помощью матрицы крайне **неэффективно**, поскольку для матричного умножения  $p \cdot M = p(1 \times 4) \cdot M(4 \times 4)$  потребуется  $(4 \cdot 4) = 16$  умножений и  $(4 \cdot 3) = 12$  сложений. А если это перемещение производится вручную, то для него будет достаточно выполнить всего три сложения:

```
x = x + xt;
y = y + yt;
z = z + zt;
w = w;
```

Отсюда следует вывод, что в процессе создания трехмерного игрового процессора можно найти **методы**, которые будут более простыми и производительными, чем матричное умножение. Однако может оказаться, что хотя эти методы и работают быстрее, они не столь понятны или не совсем подходят для аппаратного обеспечения (в котором почти во всех случаях используется преобразование с помощью матриц  $4 \times 4$ ). Поэтому при разработке игрового процессора, насколько это возможно, будут исследованы оба подхода.

Чтобы облегчить общее преобразование многоугольника или объекта с помощью матрицы  $4 \times 4$ , нам понадобится функция, в которую передаются параметры многоугольника (`PLYF4DV1`) или объекта (`OBJECT4DV1`), и в которой с помощью матричного умножения преобразуется каждая **вершина**, входящая в состав объекта. Кроме того, следует принять во внимание, что обе указанные структуры данных содержат хранилище как для локальных, так и для преобразованных вершин.

Напомним, что по окончании преобразования вершин, входящих в состав объекта, исходные данные утрачиваются, поэтому **лучше** заранее убедиться, что это преобразование действительно необходимо. Однако в некоторых случаях преобразование является неотъемлемой частью конвейера, и поэтому, возможно, понадобится сохранять копию исходных данных, оставшихся без изменения, а преобразовывать и передавать по конвейеру обработанные данные, находящиеся во вторичном хранилище. Такое место для хранения преобразованных данных предусмотрено как в структуре `POLYF4DV1`, так и в структуре `OBJECT4DV1`.

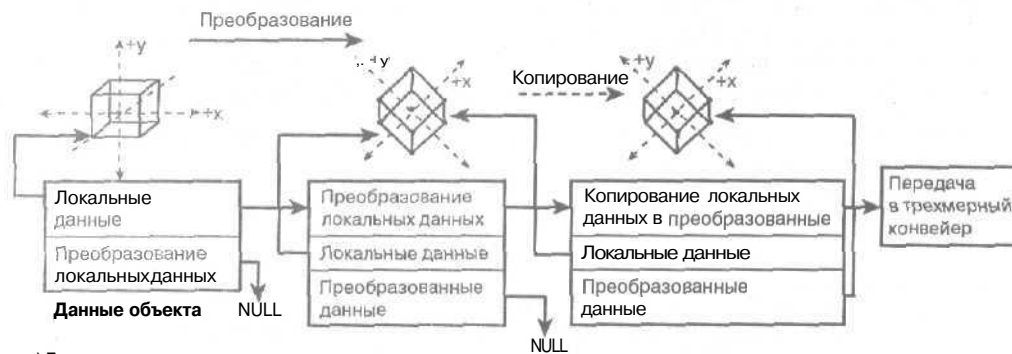
Что касается структуры `POLYF4DV1`, то вершины хранятся в ней в двух массивах.

```
POINT4D vlist[3];    // Вершины данного треугольника.
POINT4D tvlist[3];   // Вершины после преобразования
                    //(если оно необходимо)
```

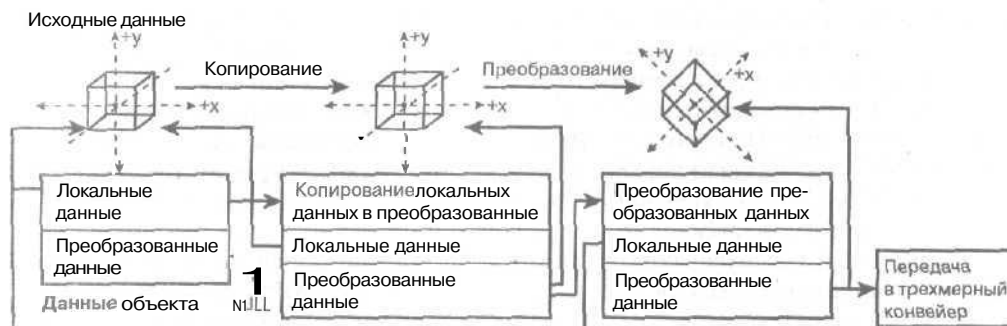
Преобразованные данные хранятся в массиве `tvlist[]`. Аналогично, в структуре `OBJECT4DV1` вершины хранятся в массивах

```
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES];
// Массив локальных вершин
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES];
// Массив преобразованных вершин
```

Теперь данные во вторичном хранилище можно стирать сколько угодно, потому что всегда есть копия исходных данных, хранящихся в локальном массиве. Однако иногда может понадобиться преобразование данных и в локальном хранилище (масштабирование или поворот, необходимые для анимации), а также занесение результатов такого преобразования в новую локальную модель. В таком случае будут модифицироваться сами локальные данные. На рис. 7.3 изображена схема двух операций, которые могут быть произведены над вершинами.



а) Деструктивная модель



б) Недеструктивная модель

Рис. 7.3. Схема деструктивного и недеструктивного преобразования параметров вершин

Итак, нам понадобится функция, предназначенная для преобразования структуры `POLYF4DV1` с помощью некоторой матрицы преобразования. Однако в большинстве случаев преобразовывать отдельный многоугольник не стоит (это было бы слишком нерационально). Лучше подождать, пока не будет сформирован полный список многоугольников. Поэтому нам нужно научиться оперировать конструкциями, подобными объекту, представляющему основной список многоугольников — который получил

имя RENDERLIST4DV1 и представляет собой не что иное, как массив, элементами которого являются объекты класса POLYF4DV1. Приведем код функции, которая выполняет преобразование списка многоугольников, заданное матрицей mt.

```
void Transform_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    MATRIX4X4_PTR mt    // Матрица преобразования
    int coord_select)    // Выбор координат для
                        // преобразования
{
    // Эта функция преобразует вершины многоугольников,
    // хранящиеся в локальном или преобразованном массиве, с
    // помощью указанной матрицы

    // Какие координаты следует преобразовывать?
    switch(coord_select)
    {
        case TRANSFORM_LOCAL_ONLY: {
            for (int poly - 0; poly < rend_list->num_polys;
                poly++)
            {
                // Запрос текущего многоугольника
                POLYF4DV1_PTR curr_poly -
                    rend_list->poly_ptrs[poly];

                // Является ли этот многоугольник
                // корректным? Многоугольник преобразуется
                // лишь в том случае, если он не отсекается
                // и не отбраковывается, если он активный и
                // видимый. Однако следует иметь в виду, что
                // концепция обратных поверхностей
                // неприменима к игровым процессорам,
                // работающим в каркасном режиме
                if ((curr_poly==NULL) ||
                    !(curr_poly->state&POLY4DV1_STATE_ACTIVE)||
                    (curr_poly->state&POLY4DV1_STATE_CLIPPED)||
                    (curr_poly->state&POLY4DV1_STATE_BACKFACE))
                    continue; // Переходим к следующему
                               // многоугольнику

                // Все в порядке, можно приступить к
                // преобразованию
                for (int vertex - 0; vertex < 3; vertex++)
                {
                    // Преобразуем параметры вершины с
                    // помощью матрицы mt
                    POINT4D presult; // Сохраняем результаты
                                    // преобразований

                    // Преобразуем координаты точки
                    Mat_Mul_VECTOR4D_4X4(
                        &curr_poly->vlist[vertex], mt
                        &presult);
                }
            }
        }
    }
}
```

```

        // Сохраняем результаты
        VECTOR4D_COPY(&curr_poly->vlist[vertex],
            &presult);
    } // for vertex
} // for poly

} break;

case TRANSFORM_TRANS_ONLY: {
    // Преобразуем каждую вершину из
    // "преобразованного" массиве списка
    // визуализации. Напомним, что предназначение
    // массива tvlist[] в том, чтобы сохранять и
    // накапливать преобразования
    for (int poly = 0; poly < rend_list->num_polys;
        poly++)
    {
        // Запрос текущего многоугольника
        POLYF4DV1_PTR curr_poly =
            rend_list->poly_ptrs[poly];
        // Является ли этот многоугольник
        // корректным? Многоугольник преобразуется
        // лишь в том случае, если он не отсекается
        // и не отбраковывается, если он активный и
        // видимый. Однако следует иметь в виду, что
        // концепция обратных поверхностей
        // неприменима к игровым процессорам,
        // работающим в каркасном режиме
        if ((curr_poly==NULL) ||
            !(curr_poly->state&POLY4DV1_STATE_ACTIVE)||
            (curr_poly->state&POLY4DV1_STATE_CLIPPED)||
            (curr_poly->state&POLY4DV1_STATE_BACKFACE))
            continue; // Переходим к следующему
                       // многоугольнику

        // Все в порядке, можно приступить к
        // преобразованию
        for (int vertex = 0; vertex < 3; vertex++)
        {
            // Преобразуем параметры вершины с
            // помощью матрицы mt
            POINT4D presult; // Сохраняем результаты
                             // преобразований

            // Преобразуем координаты точки
            Mat_Mul_VECTOR4D_4X4(
                &curr_poly->tvlist[vertex], mt
                &presult);
            // Сохраняем результаты
            VECTOR4D_COPY(&curr_poly->tvlist[vertex],
                &presult);
        } // for vertex
    }
}

```

```

    } // for poly

Ybreak;

case TRANSFORM_LOCAL_TO_TRANS: {
    // Преобразуем каждую вершину, параметры которой
    // хранятся в локальном массиве, и сохраняем
    // результаты в "преобразованном" массиве списка
    // вершин
    for(int poly = 0; poly < rend_list->num_polys;
        poly++)
    {
        // Запрос текущего многоугольника
        POLYF4DV1_PTR curr_poly =
            rend_list->poly_ptrs[poly];

        // Является ли этот многоугольник
        // корректным? Многоугольник преобразуется
        // лишь в том случае, если он не отсекается
        // и не отбраковывается, если он активный и
        // видимый. Однако следует иметь в виду, что
        // концепция обратных поверхностей
        // неприменима к игровым процессорам,
        // работающим в каркасном режиме
        if ((curr_poly==NULL) ||
            !(curr_poly->state&POLY4DV1_STATE_ACTIVE)||
            (curr_poly->state&POLY4DV1_STATE_CLIPPED)||
            (curr_poly->state&POLY4DV1_STATE_BACKFACE))
            continue; // Переходим к следующему
            // многоугольнику

        // Все в порядке, можно приступить к
        // преобразованию
        for (int vertex = 0; vertex < 3; vertex++)
        {
            // Преобразуем параметры вершины с
            // помощью матрицы mt
            Mat_Mul_VECTOR4D_4X4(
                &curr_poly->vlist[vertex], mt
                &curr_poly->tvlist[vertex]);
        } // for vertex
    } // for poly
} break;
default: break;
} // switch
} // Transform_RENDERLIST4DV1

```

Эта функция довольно проста. Она всего-навсего применяет матрицу преобразования `mt` размером 4x4 к вершинам, содержащимся в одном из массивов списка визуализации. Выбор конкретного массива осуществляется на основе флага `coord_select`. Далее, в зависимости от того, какие координаты должны быть преобразованы, происходит переход к одному из циклов, в каждом из которых вызывается функция, вычисляющая произведение матриц. Переключение вида преобразования координат контролируется параметром `coord_select` на основе приведенных ниже директив `#define`.

```
// Флаги, управляющие преобразованием
// Преобразование выполняется в списке вершин, заданном в
// локальных или мировых координатах
#define TRANSFORM_LOCAL_ONLY 0

// Преобразование выполняется в "преобразованном" списке
// вершин
#define TRANSFORM_TRANS_ONLY 1

// Преобразование выполняется в локальном списке вершин, а
// его результаты сохраняются в "преобразованном" списке
// вершин
#define TRANSFORM_LOCAL_TO_TRANS 2
```

Таким образом, возможны преобразования данных трех видов: 1) данных в локальном хранилище, 2) данных в "преобразованном" хранилище, 3) данных из локального хранилища с занесением результатов в "преобразованное" хранилище. Ничего не упущено?

Использование функции тривиально. Предположим, что в нашем распоряжении имеется список визуализации `rend_list`, заполненный многоугольниками. Допустим, что нам нужно преобразовать локальные координаты каждого многоугольника с помощью матрицы `m_trans` и сохранить результаты в "преобразованном" массиве. Это можно сделать с помощью следующего вызова:

```
Transform_RENDERLIST4DV1(&rend_list, m_trans,
    TRANSFORM_LOCAL_TO_TRANS);
```

И сразу же весь список преобразуется с помощью матрицы 4x4. Здорово, правда?

Вторая функция, которая нам понадобится, предназначена для преобразования объекта `OBJECT4DV1`. По сути, эта функция идентична предыдущей, однако она несколько проще, потому что в ней преобразуется всего лишь список вершин, а не набор многоугольников, каждый из которых включает в себя свой собственный набор вершин. Конечно же, здесь тоже следует обеспечить возможность выбора вида преобразования. С учетом сказанного функция выглядит следующим образом.

```
void Transform_OBJECT4DV1(
    OBJECT4DV1_PTR obj, // Объект, подлежащий преобразованию
    MATRIX4X4_PTR mt,   // Матрица преобразования
    int coord_select,    // Выбор преобразуемых координат
    int transform_basis) // Флаги, указывающие, нужно ли
                        // преобразовывать вектор ориентации
{
    // Эта функция преобразует параметры вершин, хранящиеся
    // в локальном или преобразованном массиве, с помощью
    // указанной матрицы

    // Какие координаты следует преобразовывать?
    switch(coord_select)
    {
        case TRANSFORM_LOCAL_ONLY: {
            // Преобразуются все вершины в локальном списке
            for (int vertex=0; vertex < obj->num_vertices;
                vertex++)
            {
                POINT4D presult; // Сохраняем результаты
                                // каждого преобразования
            }
        }
    }
}
```

```

        // Преобразуем координаты точки
        Mat_Mul_VECTOR4D_4X4(
            &obj->vlist_local[vertex], mt,
            &presult);

        // Сохраняем результаты
        VECTOR4D_COPY(&obj->vlist_local[vertex],
            &p result);
    } // forindex
} break;

case TRANSFORM_TRANS_ONLY: {
    // Преобразуем каждую вершину, параметры которой
    // хранятся в "преобразованном" массиве объекта,
    // и сохраняем результаты в том же массиве.
    // Напомним, что предназначение массива
    // vlist_trans[] в том, чтобы аккумулировать
    // преобразования
    for (int vertex=0; vertex < obj->num_vertices;
        vertex++)
    {
        POINT4D presult; // Сохраняем результаты
                        // каждого преобразования

        // Преобразуем координаты точки
        Mat_Mul_VECTOR4D_4X4(
            &obj->vlist_trans[vertex], mt,
            &presult);

        // Сохраняем результаты
        VECTOR4D_COPY(&obj->vlist_trans[vertex],
            &presult);
    } // forindex
} break;

case TRANSFORM_LOCAL_TO_TRANS: {
    // Преобразуем каждую вершину, параметры которой
    // хранятся в локальном массиве, и сохраняем
    // результаты в "преобразованном" списке вершин
    for (int vertex=0; vertex < obj->num_vertices;
        vertex++)
    {
        POINT4D presult; // Сохраняем результаты
                        // каждого преобразования

        // Преобразуем координаты точки
        Mat_Mul_VECTOR4D_4X4(
            &obj->vlist_local[vertex], mt,
            &obj->vlist_trans[vertex]);
    } // forindex
} break;

default: break;

```

```

} // switch

// Наконец, проверяем, нужно ли преобразовывать базис
// ориентации
if (transform_basis)
{
    // Изменяем ориентацию базиса объекта
    VECTOR4D vresult; // Используем для поворота каждой
    // оси вектора ориентации

    // Поворачиваем их
    Mat_Mul_VECTOR4D_4X4(&obj->ux, mt, &vresult);
    VECTOR4D_COPY(&obj->ux, &vresult);

    // Поворачиваем uy
    Mat_Mul_VECTOR4D_4X4(&obj->uy, mt, &vresult);
    VECTOR4D_COPY(&obj->uy, &vresult);

    // Поворачиваем uz
    Mat_Mul_VECTOR4D_4X4(&obj->uz, mt, &vresult);
    VECTOR4D_COPY(&obj->uz, &vresult);
} // if

} // Transform_OBJECT4DV1

```

В этой функции, как и в предыдущей, не содержится ничего, кроме кода, обрабатывающего три различных вида матричного умножения. Однако она обладает некоторой дополнительной особенностью — флагом `transform_basis`. Он нужен по следующей причине. Как видно из рис. 7.4, в процессе преобразования трехмерного объекта теряется информация о его исходной ориентации. Если известно, что **загружен** объект, для которого "верх" — это положительное направление оси *y*, а "перед" — положительное направление оси *z*, то как отследить ориентацию объекта после преобразования?

Вывод: преобразовывать нужно также базисные или локальные оси, в которых задан объект. Впоследствии с помощью этого базиса всегда можно будет определить ориентацию объекта, сравнив направление его базиса с направлением осей мировой системы координат. Это очень распространенная проблема, по поводу которой я получаю миллиарды электронных сообщений... Ну хорошо, хорошо, не **миллиарды**, а всего лишь сотни миллионов :) На рис. 7.5 изображен тот же объект после преобразования, а также его преобразованный локальный базис. Итак, флаг `transform_basis` определяет, должна ли функция преобразовывать базис объекта. Напомним, что начальный базис объекта может быть любым, однако обычно он определяется следующим образом:

$$\text{их} = \langle 1, 0, 0, 0 \rangle, \text{uy} = \langle 0, 1, 0, 0 \rangle, \text{uz} = \langle 0, 0, 1, 0 \rangle.$$

В качестве примера рассмотрим такую ситуацию: допустим, в переменную `obj` предварительно загружен объект, и **его** вместе с базисом, в котором он задан, нужно преобразовать с помощью матрицы `m_trans`. В этом случае вызов функции имеет следующий вид.

```

Transform_OBJECTDV1(&obj, m_trans,
    TRANSFORM_LOCAL_TO_TRANS, 1);

```

Пользоваться такой простой функцией очень легко.

Теперь, когда у нас есть функция преобразования общего вида, можно задать любую матрицу 4x4, а затем вызвать эту функцию. Однако напомним, что многие преобразования быстрее выполнять вручную. В следующих разделах будет показано, как реализовать

различные этапы игрового конвейера с помощью преобразований вручную и с помощью матричных методов. Оба эти способа будут применяться как к объекту `OB3ECT4DV1`, так и к списку визуализации `RENDERLIST4DV1`.

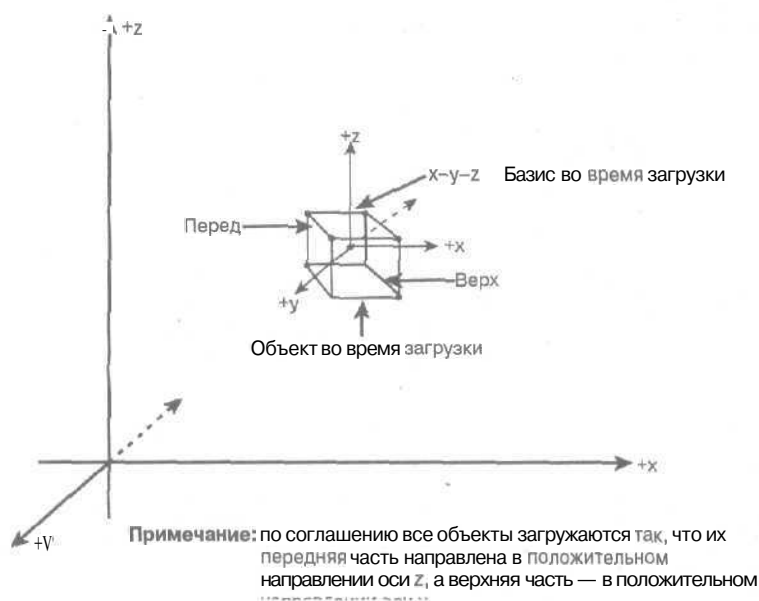


Рис. 7.4. Ориентация объекта во время загрузки

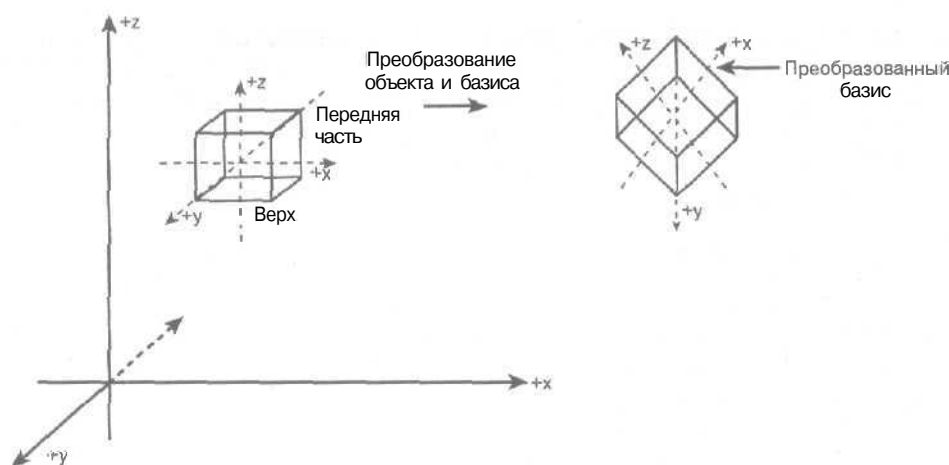


Рис. 7.5. Преобразование локального базиса для отслеживания ориентации объекта

## Преобразование из локальной системы координат в мировую

В результате перехода из локальных координат в мировые происходит преобразование локальных координат объекта (или многоугольника) в мировую систему координат (в которой задается весь мир игры; см. рис. 7.6). Таким образом, если вершина или точка за-

дана с помощью вектора  $p = \langle x, y, z, 1 \rangle$ , а смещение — с помощью вектора  $dt = \langle xt, yt, zt, 1 \rangle$ , то уравнение перехода из локальных координат в мировые имеет вид:  
 $p' = p + dt$ .  
 При этом компонент  $w = 1$  остается неизменным.

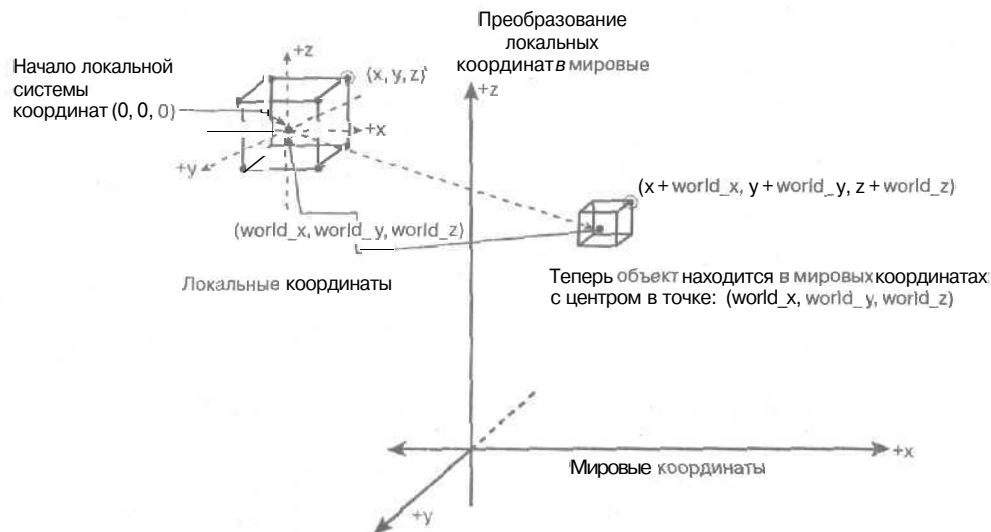


Рис. 7.6. Преобразование из локальной в мировую систему координат

### Преобразование объектов из локальных координат в мировые

Функция, осуществляющая преобразование объекта `OBJECT4DV1` из локальной системы координат в мировую, просто переносит вершины этого объекта, параметры которых хранятся в массиве `local_vlist[]`, на величину `world_pos` и сохраняет результаты в массиве `trans_vlist[]`. Ниже приведен код функции, в которой это преобразование выполняется вручную.

```
void Model_To_World_OBJECT4DV1(
    OBJECT4DV1_PTR obj,
    int coord_select = TRANSFORM_LOCAL_TO_TRANS)
{
    // ПРИМЕЧАНИЕ: матрица не используется.
    // Эта функция преобразует модель указанного объекта,
    // заданную в локальных координатах, в мировую систему
    // координат. Результаты заносятся в преобразованный
    // список вершин (vlist_trans) объекта

    // По всем элементам списка вершин производятся
    // итерации, в которых локальные координаты вершин
    // преобразуются в мировые координаты. Величина смещения
    // задается с помощью переменной world_pos, а результаты
    // заносятся в массив vlist_trans[]

    if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
    {
        for (int vertex=0; vertex < obj->num_vertices;

```

```

        vertex++)
    {
        // Переносим вершину
        VECTOR4D_Add(&obj->vlist_local[vertex],
                    &obj->world_pos,
                    &obj->vlist_trans[vertex]);
    } // for vertex
} // if local
else // TRANSFORM_TRANS_ONLY
{
    for (int vertex=0; vertex < obj->num_vertices;
        vertex++)
    {
        // Переносим вершину
        VECTOR4D_Add(&obj->vlist_trans[vertex],
                    &obj->world_pos,
                    &obj->vlist_trans[vertex]);
    } // for vertex
} // else trans

} // Model_To_World_OBJECT4DV1

```

Если в объект `OBJECT4DV1` предварительно загружен куб с единичной стороной, а переменной `world_pos` присвоено значение (100,200,400), то весь каркас можно транслировать с помощью одного вызова

```
Model_To_World_OBJECT4DV1(&cube);
```

Заметим, что в функции возможен выбор используемых координат. Этот выбор нужен в тех ситуациях, когда преобразование модели осуществляется уже после того, как ее координаты претерпели преобразование. В этом случае для преобразования должны использоваться не исходная, а преобразованная система координат модели. Как обычно, это контролируется с помощью параметра `coord_select`.

А теперь, если хотите, мы можем выполнить это же преобразование и с помощью матрицы, хотя это будет напрасной тратой ресурсов. Тем не менее, стоит взглянуть и на этот вариант, так как аппаратные решения любят использовать матрицы. Функция, которая конструирует матрицу преобразования, получила название `Build_Model_To_World_MATRIX4X4()`, а ее код приведен ниже.

```
void Build_Model_To_World_MATRIX4X4(VECTOR4D_PTR vpos,
                                     MATRIX4X4_PTR m)
```

```

{
    // Эта функция конструирует общую матрицу перехода из
    // локальных координат в мировые. Это матрица, с помощью
    // которой начало координат сдвигается в точку vpos

```

```

    Mat_Init_4X4(m, 1, 0, 0, 0,
                 0, 1, 0, 0,
                 0, 0, 1, 0,
                 vpos->x, vpos->y, vpos->z, 1);

```

Как видите, в функцию передаются координаты точки, в которую нужно перенести начало координат, а также переменная-хранилище, куда нужно занести элементы полученной матрицы преобразования. Конечно же, эту матрицу можно использовать не ТОЛЬ-

ко для перехода из локальных координат в мировые, но и для других преобразований. Если преобразование локальных координат нужно выполнить с **помощью матрицы**, то можно сначала вызвать функцию `Build_Model_To_World_MATRIX4X4()`, чтобы сконструировать матрицу, а затем — запустить функцию преобразования.

```
MATRIX4X4 mt; // Переменная для хранения матрицы
VECTOR4D pos = {100,200,300, 1};
```

```
Build_Model_To_World_MATRIX4X4(&pos, &mt);
```

```
Transform_OBJECT4DV1(&obj, m_trans,
    TRANSFORM_LOCAL_TO_TRANS,1);
```

Вот и **все**, что я хотел рассказать о преобразовании объектов класса `OBJECT4DV1` из локальных координат в мировые.

**НА ЗАМЕТКУ**

Я знаю, что **все** это тривиально, но зато мы приобрели **полезные** навыки.

## Преобразование списков визуализации из локальных координат в мировые

В большинстве случаев многоугольники, из которых состоит объект, помещаются в список визуализации уже после того, как объект преобразован из локальных координат в мировые. При этом может не понадобиться преобразование всего списка визуализации из локальных координат в мировые. Однако если вы не хотите использовать объект класса `OBJECT4DV1`, а загружаете многоугольники непосредственно в список `RENDERLIST4DV1`, представляющий большой каркас (например, это может быть ландшафт), то без этого шага не обойтись, если только каркас задан не в мировых координатах. Приведем код функции, в которой вручную, без использования матриц, производится преобразование списка `RENDERLIST4DV1` из локальных координат в мировые.

```
void Model_To_World_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    POINT4D_PTR world_pos,
    int coord_select - TRANSFORM_LOCAL_TO_TRANS)
1
// ПРИМЕЧАНИЕ: матрица не используется.
// Эта функция преобразует указанный список
// визуализации, заданный в локальных координатах, в
// мировую систему координат. Результаты заносятся в
// преобразованный список вершин (tvlist) списка
// визуализации

// По всем элементам списка вершин производятся
// итерации, в которых локальные координаты модели
// преобразуются в мировые координаты путем смещения на
// величину world_pos. Результаты заносятся в массив
// vlist_trans[]

if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
{
    for(int poly = 0; poly < rend_list->num_polys;
        poly++)
    {

```

```

// Запрос текущего многоугольника
POLYF4DV1_PTR curr_poly =
    rend_list->poly_ptrs[poly];

// Многоугольник преобразуется лишь в том
// случае, если он не отсекается и не
// отбраковывается, если он активный и видимый.
// Однако следует иметь в виду, что концепция
// обратных поверхностей неприменима к игровым
// процессорам, работающим в каркасном режиме
if ((curr_poly==NULL) ||
    !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // Переходим к следующему
               // многоугольнику

// Все в порядке, можно приступить к
// преобразованию
for (int vertex = 0; vertex < 3; vertex++)
{
    // Преобразуем параметры вершины
    VECTOR4D_Add(&curr_poly->vlist[vertex],
        world_pos,
        &curr_poly->tvlist[vertex]);
} // for vertex
} // for poly
} // if Local
else // TRANSFORM_TRANS_ONLY
{
    for (int poly = 0; poly < rend_list->num_polys;
        poly++)
    {
        // Запрашиваем текущий многоугольник
        POLYF4DV1_PTR curr_poly =
            rend_list->poly_ptrs[poly];

        // Многоугольник преобразуется лишь в том
        // случае, если он не отсекается и не
        // отбраковывается, если он активный и видимый.
        // Однако следует иметь в виду, что концепция
        // обратных поверхностей неприменима к игровым
        // процессорам, работающим в каркасном режиме
        if ((curr_poly==NULL) ||
            !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
            (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
            (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
            continue; // Переходим к следующему
                       // многоугольнику

        for (int vertex = 0; vertex < 3; vertex++)
        {
            // Переносим вершину.

```

```

        VECTOR4D_Add(&curr_poly->tvlist[vertex],
                    world_pos,
                    &curr_poly->tvlist[vertex]);
    } // for vertex

} // for poly

} // else

} // Model_To_World_RENDERLIST4DV1

```

Таким образом, если имеется список визуализации `rend_list` и его нужно преобразовать из локальных координат в мировые путем переноса всех геометрических элементов из их локальных позиций в мировые с использованием вектора `world_pos`, это можно выполнить следующим образом.

```

Model_To_World_RENDERLIST4DV1(&rend_list, &world_pos,
                               TRANSFORM_LOCAL_TO_TRANS);

```

Это очень эффективная функция. Если то же самое нужно сделать с помощью матрицы, то сначала следует сконструировать саму матрицу преобразования.

```

MATRIX4X4 mt; // Матрица-хранилище
VECTOR4D pos = {100,200,300, 1};

```

```

Build_Model_To_World_MATRIX4X4(&pos, &mt);

```

Затем вызывается функция, выполняющая общее преобразование списка визуализации.

```

Transform_RENDERLIST4DV1(&rend_list, &mt,
                          TRANSFORM_LOCAL_TO_TRANS);

```

**НА ЗАМЕТКУ**

Заметим, что использование **такого** метода **приводит к напрасному расходу времени** процессора **из-за** лишних операций, необходимых для матричного **умножения координат** каждой вершины.

## Эйлерова модель камеры

На самом деле моделирование камеры не выделяется как отдельный этап трехмерного игрового конвейера, но сейчас пришло время поговорить об этом вопросе, поскольку последующие этапы конвейера основаны на используемой модели камеры. Напомним, что модель камеры, которая используется в нашей первой системе, называется *эйлеровой* (Eulerian). Это означает, что камера задается с помощью положения и углов поворота (или углов Эйлера), задающих ее ориентацию (рис. 7.7). Кроме того, мы собираемся поддерживать UVN-камеру, для которой требуется только задать направление на целевую точку и векторы *и*, *у*, *н*. Однако подробнее мы поговорим об этом позже.

Какая бы модель камеры не применялась, необходимо задать расстояние до наблюдаемой точки, поле обзора, ближнюю и дальнюю плоскости отсечения, а также плоскость определенной ширины и высоты, представляющую конечное изображение, которое будет выведено на экран на определенном этапе, а также указать размеры экрана, представляющего окно растеризации. Все эти данные должны учитываться в модели камеры, поскольку они определяют видимое пространство, прилегающее к камере и задающее ее свойства. Объекты, попадающие в поле зрения, определенным образом преобразуются и отображаются на экране.

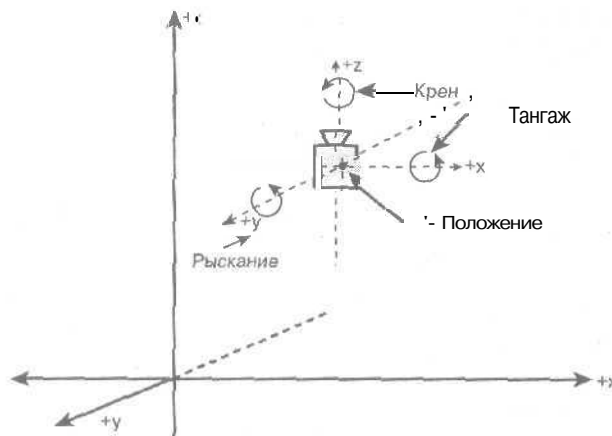


Рис. 7.7. Эйлерова модель камеры

Подводя итоги по данной теме, я сел за компьютер и написал структуры, представляющие простую камеру. На самом деле, скорее всего, можно было бы обойтись углами, координатами, расстояниями до ближней и дальней плоскостей отсечения и еще парой параметров, но мне хочется приучить вас **заблаговременно** думать о том, какая информация может понадобиться камере, даже если пока что эта информация не используется. Увы, мне пришлось ввести несколько параметров, которые в этой главе окажутся не нужны; в первую очередь это касается величин, имеющих отношение к отсечению. Итак, приведем структуру для описания камеры.

// Камера; версия 1.

typedef struct CAM4DV1\_TYP

```
{
    int state;           // Состояние камеры
    int attr;           // Атрибуты камеры

    POINT4D pos;        // Положение камеры в мировых координатах,
                        // использующееся в обеих моделях камеры

    VECTOR4D dir;       // Углы Эйлера или направление в
                        // UVN-моделях

    VECTOR4D и;         // Дополнительные векторы, с помощью
    VECTOR4D v;         // которых отслеживается ориентация камеры
    VECTOR4D п;         // для более сложных UVN-моделей камеры

    PQINT4D target;     // Целевая точка, на которую "смотрит"
                        // камера в модели UVN

    float view_dist_h;  // Расстояния обзора по горизонтали
    float view_dist_v;  // и вертикали.

    float fov;         // Поле обзора по горизонтальной и
                        // вертикальной осям
}
```

```

// Плоскости отсечения в трехмерном пространстве. Если
// область видимости не имеет форму прямоугольного
// параллелепипеда, необходимо применять отсечение
// общего вида в трехмерном пространстве
float near_clip_z; // Ближняя плоскость отсечения
float far_clip_z; // Дальняя плоскость отсечения

PLANE3D rt_clip_plane; // Правая плоскость отсечения
PLANE3D lt_clip_plane; // Левая плоскость отсечения
PLANE3D tp_clip_plane; // Верхняя плоскость отсечения
PLANE3D bt_clip_plane; // Нижняя плоскость отсечения

float viewplane_width; // Ширина и высота области, на
// которую производится
// проецирование
float viewplane_height; // Обычно задается размер 2x2 для
// ортонормированной проекции или
// точно такой же размер для
// демонстрационного окна или
// окна экрана

// Следует запомнить, что экран и демонстрационное
// окно - это синонимы
float viewport_width; // Размер экрана
float viewport_height;
float viewport_center_x; // Центр экрана (положение
// конечного изображения)
float viewport_center_y;

// Соотношение геометрических размеров
float aspect_ratio; // Форматное отношение

// Данные матрицы могут и не понадобиться; это зависит
// от метода преобразования. Например, они не нужны для
// аксонометрического преобразования, выполняемого
// вручную или для перехода в систему координат экрана.
// Однако наличие этих матриц придает большую гибкость

MATRIX4X4 mcam; // Матрица перехода из координат матрицы
// в координаты камеры
MATRIX4X4 mper; // Матрица перехода из координат камеры
// в аксонометрические координаты
MATRIX4X4 mscr; // Матрица перехода из аксонометрических
// координат в экранные

} CAM4DV1, *CAM4DV1_PTR;

```

Несмотря на наличие в приведенной структуре комментариев, еще раз рассмотрим каждый ее элемент во избежание недоразумений.

state — состояние камеры. В этом параметре нет ничего сверхъестественного.

attr — определенные атрибуты камеры. В данный момент имеются атрибуты только двух типов: CAM\_MODEL\_EULER и CAM\_MODEL\_UVN.

pos — положение камеры в мировых координатах.

`dir` — в эйлеровой модели камеры направление осей, а в модели **UVN** — компонента  $x$  соответствует углу возвышения в сферической системе координат, а компонента  $y$  — это угол направления.

$u, v, n$  — координаты вектора, отслеживающего ориентацию камеры для более сложной **UVN-модели**, о которой пойдет речь в последующих разделах главы.

`target` — точка, на которую направлена камера в модели **UVN**.

`view_dist_h, view_dist_v` — расстояния по горизонтали и вертикали, используемые при аксонометрическом преобразовании.

`fov` — угол обзора по горизонтальной и вертикальной осям.

`near_clip_z, far_clip_z` — ближняя и дальняя плоскости отсечения, перпендикулярные оси  $z$ .

`rt_clip_plane, lt_clip_plane, tp_clip_plane, bt_clip_plane` — правая, левая, верхняя и нижняя плоскости отсечения в трехмерном пространстве. Если угол обзора равен  $90^\circ$ , эти плоскости составляют угол  $45^\circ$  с плоскостями  $xy$ ,  $yz$  и  $zx$ , а их уравнения имеют вид  $x = \pm z$  и  $y = \pm z$ . Однако угол обзора может быть произвольным, поэтому процедуру отсечения в трехмерном пространстве необходимо обобщить.

`viewplane_width, viewplane_height` — размер плоскости, на которую выполняется проецирование. Для нормированной проекции это размер  $2 \times 2$ , а для тех случаев, когда аксонометрическая проекция объединяется с переходом к экранным координатам (в целях повышения производительности программы), — этот размер совпадает с размером окна на экране.

`viewport_width, viewport_height` — выраженный в пикселях размер конечной поверхности, для которой выполняется растеризация.

`viewport_center_x, viewport_center_y` — центр экрана.

`aspect_ratio` — отношение `viewport_width/viewport_height`.

`mcam` — хранилище для параметров, полученных в результате перехода в координаты камеры.

`mper` — хранилище для параметров, полученных в результате перехода из координат камеры в аксонометрические координаты.

`mscr` — хранилище для параметров, полученных в результате перехода из аксонометрических координат в экранные.

Ух, как много всего! Однако будьте уверены, вскоре все эти параметры понадобятся. Предназначение представленной структуры — хранить все данные, имеющие отношение к камере, но в ней много и выводимых данных, которые не хотелось бы определять или вычислять самостоятельно. Поэтому нам нужна функция, способная принять набор простых входных величин и построить модель камеры. Начнем с прототипа такой функции.

```
void Init_CAM4DV1(CAM4DV1_PTR cam, // Объект-камера
    int cam_attr, // Атрибуты камеры
    POINT4D_PTR cam_pos, // Начальное положение
    // камеры
    VECTOR4D_PTR cam_dir, // Начальная ориентация
    // камеры
    " POINT4D_PTR cam_target // Начальная целевая
    // точка в модели UVN
    float near_clip_z, // Ближняя и дальняя
    float far_clip_z, // плоскости отсечения
    float fov, // Угол обзора в градусах
    float viewport_width, // Окончательные размеры
    float viewport_height); // экрана
```

В эту функцию передается тип камеры, ее положение, ориентация, расстояние от камеры до плоскостей отсечения, размеры поля обзора и экрана, после чего она определяет все необходимые параметры камеры. Единственная проблема заключается в том, что многие из этих параметров взаимосвязаны, поэтому одни накладывают ограничения на другие и наоборот.

Поэтому функция написана так, что она во **всех** случаях создает поле зрения в ширину  $[-1, 1]$  и в высоту  $\{-1/\text{ar}, 1/\text{ar}\}$ . Об этом нужно договориться заранее, потому что в процессе аксонометрического преобразования или отображения на экран **нужно** знать, что именно проецируется. Данное соглашение **существенно** все упрощает.

В большинстве случаев будет использоваться поле зрения с углом обзора, равным  $90^\circ$ , из чего следует, что фокусное расстояние равно 1.0. Однако мне хотелось придать дополнительные возможности функции, задающей параметры камеры, чтобы можно было поэкспериментировать с углами обзора, отличными от прямых. Наконец, было решено, что в этой функции **НЕ** следует вычислять матрицу камеры в процессе инициализации. Эту матрицу нужно создавать позже, с помощью вспомогательной функции, работающей либо с эйлеровой моделью камеры, либо с моделью UVN. А теперь рассмотрим модель UVN.

## Модель UVN камеры

Эйлерова модель камеры подходит для многих случаев. Однако она довольно ограничена и работает не совсем так, как мы **предполагаем**, когда задаем углы обзора. Более естественной является модель UVN. Работать с ней намного легче, поэтому мы сейчас займемся рассмотрением этой модели, а также матрицы преобразования камеры. Стандартная система UVN показана на рис. 7.8.

По сути, единственное отличие эйлеровой камеры от камеры UVN — в способе задания ориентации: вместо углов используются векторы, с помощью которых определяется ориентация (или базис) системы координат камеры:

$$\text{UVN} = \{X: \langle u_x, u_y, u_z \rangle, Y: \langle v_x, v_y, v_z \rangle, Z: \langle n_x, n_y, n_z \rangle\}.$$

Например, в приведенных выше обозначениях базис стандартной левосторонней системы XYZ имел бы такой вид:

$$\text{XYZ} = \{X: \langle 1, 0, 0 \rangle, Y: \langle 0, 1, 0 \rangle, Z: \langle 0, 0, 1 \rangle\}.$$

Система UVN обладает такими же **свойствами**, что и система координат XYZ. Ее оси взаимно перпендикулярны, или **ортогональны**, а векторы — линейно независимы и имеют единичную длину.

В систему UVN также входит так называемая **целевая точка** (target point, look at point), которая представляет собой просто точку, на которую нацелена камера. Кроме того, конечно же, камера характеризуется положением в пространстве, которое называется **опорной точкой** (view reference point, VRP).

А теперь поговорим о векторах UVN и о том, что они обозначают, хотя при этом и нарушим немного порядок изложения.

**n** — этот вектор аналогичен оси z в системе UVN; он соединяет опорную точку с целевой. Другими словами,  $\mathbf{n} = \langle \text{target} - \text{VRP} \rangle$ . Конечно же, вектор **n** должен быть **единичным**, поэтому на определенном этапе его нужно будет нормировать.

**v** — это так называемый "вектор вверх". Вначале он идентичен вектору, направленному вдоль оси y:  $\langle 0, 1, 0 \rangle$ . Однако эти координаты вектора лишь временные и нужны только для того, чтобы вычислить вектор **i**, направленный "вправо". После этого вычисляются новые координаты вектора **v**.

**i** — это так называемый "вектор вправо". Он вычисляется с помощью векторного произведения векторов **v** и **n**:  $\mathbf{i} = (\mathbf{v} \times \mathbf{n})$ . Вот здесь-то и оказываются полезными временные координаты вектора **v**. Как только становятся известны координаты вектора **i**, **повторно** вычисляется вектор **v** по формуле  $\mathbf{v} = (\mathbf{n} \times \mathbf{i})$ .

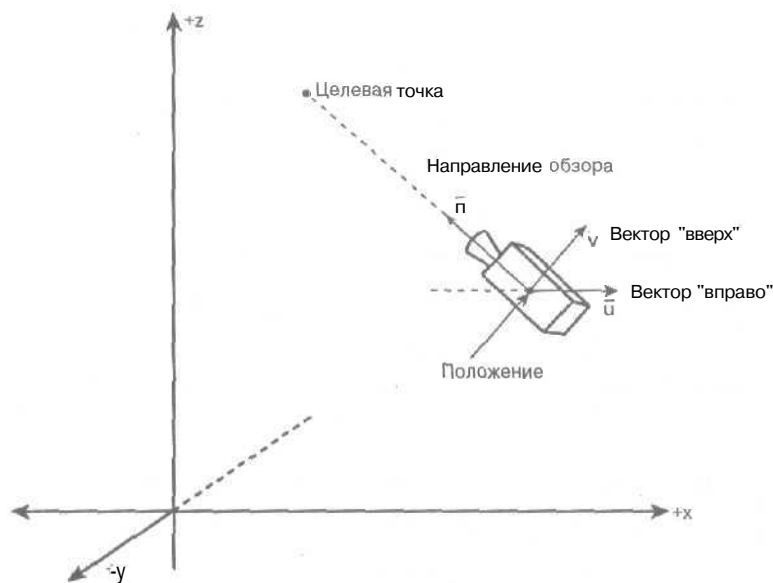


Рис. 7.8. Модель UVN-камеры

НА ЗАМЕТКУ

Конечно же, на определенном этапе все векторы следует нормировать.

Предположим, что мы вычислили векторы  $u$ ,  $v$  и  $p$  системы UVN. Теперь осталось понять, какой в них заложен смысл.

На самом деле все очень просто. Система UVN — это, по сути, новая ориентация системы XYZ, в которую будут преобразованы все геометрические элементы. Другими словами, это базис новой системы координат. Таким образом, нужно построить матрицу перехода, с помощью которой можно было бы произвести отображение одной системы координат в другую. Предположим, что камера UVN находится в некоторой точке пространства и направлена на определенную целевую точку. Какой должна при этом быть матрица перехода в систему координат камеры? Давайте сначала забудем о положении камеры, ведь мы все знаем, что трансляция выполняется просто. Трудная часть — поворот...

### Вычисление матрицы UVN

Оказывается, преобразование координат точки  $p(x, y, z)$  из стандартной системы координат XYZ в систему UVN сводится к обычному скалярному произведению вектора  $p$  на векторы  $u$ ,  $v$  и  $p$ . Другими словами, чтобы определить координаты вектора в новой системе координат, нужно спроецировать его на каждую из осей этой системы. Таким образом, матрица преобразования  $4 \times 4$  имеет вид:

$$M_{uvn} = \begin{bmatrix} ux & vx & px & 0 \\ uy & vy & py & 0 \\ uz & vz & pz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ловко, правда? Столбцы матрицы представляют собой векторы  $u$ ,  $v$  и  $p$ , соответственно. Предположим, что точку  $p(x, y, z, 1)$  нужно преобразовать с помощью матрицы в новую систему координат. Для преобразованных координат имеем такие соотношения:

$$\begin{aligned}
p' &= p \cdot M_{uvn} = \\
&= [x \ y \ z \ 1] \cdot \begin{bmatrix} ux & vx & px & 0 \\ uy & vy & py & 0 \\ uz & vz & pz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\
&= p'((x \cdot ux + y \cdot uy + z \cdot uz), (x \cdot vx + y \cdot vy + z \cdot vz), (x \cdot px + y \cdot py + z \cdot pz), 1),
\end{aligned}$$

что на самом деле представляет собой скалярное произведение рассматриваемых векторов;

$$p' = (p \cdot u, p \cdot v, p \cdot n).$$

Осталось решить загадку о положении камеры, но это легко. Известно, что любое преобразование камеры можно представить в виде ее последовательных сдвига и вращения:

$$T_{cam}^{-1} \cdot R_{cam}^{-1},$$

где  $R_{cam}^{-1}$  — это один поворот или сочетание нескольких поворотов,

**НА ЗАМЕТКУ**

Символ обратного преобразования должен послужить напоминанием о том, что исходная матрица переносит камеру в заданную точку. Чтобы преобразовать координаты всех окружающих предметов в систему координат камеры, нужно выполнить обратное преобразование. Аналогичные рассуждения справедливы и для матрицы поворота.

У нас уже есть матрица поворота, которая названа  $M_{uvn}$ , так что осталось найти матрицу переноса. Предположим, что камера находится в точке с координатами  $cam\_pos = (cam\_x, cam\_y, cam\_z)$ . Тогда матрица переноса имеет такой вид:

$$T_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}.$$

В результате мы приходим к тому, что конечный вид матрицы преобразования для камеры UVN следующий:

$$\begin{aligned}
T_{uvn} &= T_{cam}^{-1} \cdot M_{uvn} = \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix} \cdot \begin{bmatrix} ux & vx & px & 0 \\ uy & vy & py & 0 \\ uz & vz & pz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\
&= \begin{bmatrix} ux & vx & px & 0 \\ uy & vy & py & 0 \\ uz & vz & pz & 0 \\ -(cam\_pos \cdot u) & -(cam\_pos \cdot v) & -(cam\_pos \cdot n) & 1 \end{bmatrix}.
\end{aligned}$$

Элементы последней строки — это множители  $tx$ ,  $ty$  и  $tz$ , отвечающие за перенос. Они представляют собой взятое с противоположным знаком скалярное произведение радиус-вектора камеры на векторы  $u$ ,  $v$  и  $n$ , соответственно.

Вот и все, что я хотел сказать по поводу системы UVN. А теперь кратко рассмотрим (а позже займемся реализацией), как на практике описывается система UVN и как в ней производятся расчеты.

В системе UVN камера моделируется как обычно, с помощью указания ее положения, однако отличительная особенность этой модели состоит в том, что у камеры есть целевая точка, на которую она направлена. Кроме того, в модели UVN камера представляется не в виде ряда расположенных в определенном порядке последовательных поворотов, параллельных каждой из осей, а с помощью вектора  $\mathbf{n}$ , направленного на целевую точку, вектора  $\mathbf{v}$ , направленного "вверх", и, наконец, вектора  $\mathbf{u}$ , направленного "вправо". Такое представление коррелирует с осями стандартной декартовой системы XYZ:

$$\mathbf{U} \rightarrow \mathbf{X},$$

$$\mathbf{V} \rightarrow \mathbf{Y},$$

$$\mathbf{N} \rightarrow \mathbf{Z}.$$

Дадим строгое описание действий по вычислению векторов  $\mathbf{u}$ ,  $\mathbf{v}$  и  $\mathbf{n}$ .

1.  $\mathbf{n} = \langle \text{целевая точка} - \text{положение камеры} \rangle$ .
2. Примем  $\mathbf{v} = \langle 0, 1, 0 \rangle$ .
3.  $\mathbf{u} = (\mathbf{v} \times \mathbf{n})$
4.  $\mathbf{v} = (\mathbf{n} \times \mathbf{u})$
5. (Необязательно): нормируем векторы  $\mathbf{u}$ ,  $\mathbf{v}$  и  $\mathbf{n}$  путем деления каждого из них на длину (если это не сделано на предшествующих этапах).

### Практическая реализация системы UVN

Как видим, система UVN — достаточно полезная вещь, и в большой степени она соответствует тому, как камера используется в реальности. Камера помещается в определенную точку, а затем направляется на определенную цель (объект съемки, см. рис. 7.8). Единственная проблема заключается в том, что часто неизвестны точные координаты точки, на которую направлена камера, а задано только общее направление. Другими словами, зачастую больше подходит эйлерова модель, но, как вы уже поняли, она не всегда ведет себя, как ожидается, и ей присущи такие недостатки, как блокировка подвески и другие. Поэтому, если известно, куда нужно направить камеру, векторы  $\mathbf{u}$ ,  $\mathbf{v}$  и  $\mathbf{n}$  можно вычислить непосредственно, а потом работать с ними. Но есть и другие модели камеры — например, основанная на сферической системе координат, в которой задается положение камеры, а затем — углы направления и возвышения (рис. 7.9), после чего опять же можно математически вычислить векторы  $\mathbf{u}$ ,  $\mathbf{v}$  и  $\mathbf{n}$ . Это совсем не так сложно, как кажется. Рассмотрим рис. 7.9, на котором проиллюстрирована описанная схема.

Для начала, чтобы упростить рассуждения, предположим, что камера находится в начале координат (и окружена сферической оболочкой единичного радиуса). Очевидно, для перемещения камеры в произвольную точку пространства достаточно будет произвести операцию переноса. Как видно из рисунка, угол направления  $\theta$  — отсчитываемый против часовой стрелки угол поворота вокруг положительного направления оси  $y$ . В соответствии с принятым соглашением, нулевому углу соответствует направление вдоль положительного направления оси  $z$ . Далее, угол возвышения  $\phi$  — угол между базовой плоскостью  $xz$  и вектором направления (см. рис. 7.9). В сферических координатах точка обычно обозначается как  $\mathbf{p}(\rho, \phi, \theta)$ . Однако в главе 4, "Запутанный мир математики", предполагалось использование правой системы координат, а также других соглашений относительно углов. Поэтому при использовании формул из этой главы следует проявлять осторожность, и их нужно слегка модифицировать, как описано ниже.

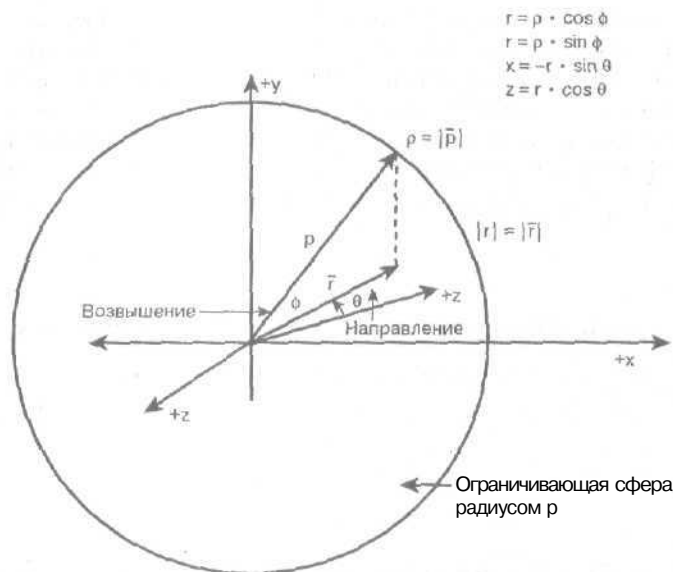


Рис. 7.9. Схема UVN-камеры в сферических координатах

Введем следующие обозначения для камеры, положение которой задано вектором  $p(p, \phi, \theta)$ :

- $p$  — целевой вектор;
- $p$  — длина целевого вектора, которая в данном случае равна 1.0, поскольку камера ограничена единичной сферой;
- $\theta$  — угол направления;
- $\phi$  — угол возвышения;
- $r$  — длина проекции целевого вектора на плоскость  $xz$ .

Тогда можно записать следующие соотношения.

#### Уравнение 7.1 . Вычисление целевого вектора в системе UVN с помощью сферических координат

$$\begin{aligned} r &= p \cdot \cos \phi, \\ x &= -r \cdot \sin \theta, \\ y &= p \cdot \sin \phi, \\ z &= r \cdot \cos \theta. \end{aligned}$$

Внимательно присмотревшись к координатам, которые генерирует функция SPHERICAL3D\_To\_POINT3D(), нетрудно убедиться, что они основаны на правой системе координат. Чтобы отобразить их в нашу систему, следует выполнить такие преобразования:

$$\begin{aligned} x &= -y, \\ y &= z, \\ z &= x. \end{aligned}$$

Вычисленные значения представляют собой компоненты вектора  $p(x, y, z)$ , задающего направление на целевую точку. Таким образом, все параметры камеры в модели Эйлера и модели UVN можно задать с помощью углов направления и возвышения,

а также положения камеры, поскольку на основе этих величин вычисляются все остальные параметры.

Обладая знаниями об эйлеровой модели камеры и системе UVN, можно создать любую другую модель камеры, какая только потребуется. Скоро будет приведен код, в котором задаются параметры камеры, а также вызываются различные функции, создающие матрицы преобразования в систему координат камеры для обеих систем.

## Инициализация камеры

Как уже упоминалось, в функции, моделирующей камеру, матрица перехода в систему координат камеры не задается (это делается с помощью вспомогательной функции), но в ней устанавливаются все остальные параметры, имеющие отношение к полю зрения, плоскостям отсечения и т.д. Приведем код этой функции.

```
void Init_CAM4DVI(
    CAM4DV1_PTR cam,           // Объект камеры
    int cam_attr,              // Атрибуты камеры
    POINT4D_PTR cam_pos,       // Начальное положение камеры
    VECTOR4D_PTR cam_dir,      // Начальное направление камеры
    POINT4D_PTR cam_target,    // Начальная целевая точка в
                                // модели UVN
    float near_clip_z,         // Ближняя и дальняя
    float far_clip_z,          // плоскости отсечения
    float fov,                 // Угол обзора в градусах
    float viewport_width,      // Окончательный размер экрана
    float viewport_height)
{
    // Эта функция инициализирует объект cam, представляющий
    // камеру. В ней не выполняется серьезная проверка на
    // наличие ошибок. Кроме того, количество необходимых
    // параметров сведено к минимуму

    // Сначала задаются простые параметры
    cam->attr = cam_attr;           // Атрибуты камеры

    VECTOR4D_COPY(&cam->pos, cam_pos); // Положение
    VECTOR4D_COPY(&cam->dir, cam_dir);  // Вектор направления
                                        // или углы для эйлеровой камеры

    // Для камеры в модели UVN
    VECTOR4D_INITXYZ(&cam->u, 1,0,0); // Вдоль оси x
    VECTOR4D_INITXYZ(&cam->v, 0,1,0); // Вдоль оси y
    VECTOR4D_INITXYZ(&cam->n, 0,0,1); // Вдоль оси z

    if (cam_target != NULL)
        VECTOR4D_COPY(&cam->target, cam_target);
        // Целевая точка в модели UVN
    else
        VECTOR4D_ZERO(&cam->target);

    // Ближняя плоскость отсечения
    cam->near_clip_z = near_clip_z;
    // Дальняя плоскость отсечения
    cam->far_clip_z = far_clip_z;
```

```

cam->viewport_width - viewport_width; // Размеры экрана
cam->viewport_height - viewport_height;

cam->viewport_center_x=(viewport_width-1)/2; // Центр
cam->viewport_center_y=(viewport_height-1)/2; // экрана

cam->aspect_ratio - (float)viewport_width/
                    (float)viewport_height;

// В качестве всех матриц указываем единичную матрицу
MAT_IDENTITY_4X4(&cam->mcam);
MAT_IDENTITY_4X4(&cam->mper);
MAT_IDENTITY_4X4(&cam->mscr);

// Задаем независимые переменные
cam->fov = fov;

// Задаем размеры поля наблюдения;
// они будут равны 2 x (2/ar)
cam->viewplane_width - 2.0;
cam->viewplane_height - 2.0/cam->aspect_ratio;

// Теперь известно значение переменной fov, а также
// размеры поля обзора. Подставляем эти величины в
// формулу и находим параметры, относящиеся к расстоянию
// от точки наблюдения
float tan_fov_div2 = tan(DI_TO_RAD(fov/2));

cam->view_dist =
    (0.5)*(cam->viewplane_width)*tan_fov_div2;

// Сначала проверяем, равен ли угол обзора 90, поскольку
// это самый легкий случай :)
if (fov == 90.0)
{
    // Задаем плоскости отсечения - для угла обзора
    // в 90° это очень просто!
    POINT3D pt_origin; // Указатель на плоскость
    VECTOR3D_INITXYZ(&pt_origin,0,0,0);

    VECTOR3D vn; // Нормаль к плоскости

    // Правая плоскость отсечения
    VECTOR3D_INITXYZ(&vn,1,0,-1); // Плоскость x=z
    PLANE3D_Init(&cam->rt_clip_plane,&pt_origin,&vn,1);

    // Левая плоскость отсечения
    VECTOR3D_INITXYZ(&vn,-1,0,-1); // Плоскость -x=z
    PLANE3D_Init(&cam->lt_clip_plane,&pt_origin,&vn,1);

    // Верхняя плоскость отсечения
    VECTOR3D_INITXYZ(&vn,0,1,-1); // Плоскость y=z

```

```

PLANE3D_Init(&cam->tp_clip_plane,&pt_origin,&vn,1);

// Нижняя плоскость отсечения
VECTOR3D_INITXYZ(&vn,0,-1,-1); // Плоскость -y-z
PLANE3D_Init(&cam->bt_clip_plane,&pt_origin,&vn,1);
} // if d=1
else
{
    // Вычисляем параметры произвольных плоскостей
    // отсечения
    POINT3D pt_origin; // Указатель на плоскость
    VECTOR3D_INITXYZ(&pt_origin,0,0,0);

    VECTOR3D vn; // Нормаль к плоскости

    // Поскольку угол обзора отличен от прямого,
    // вычисление нормалей представляет определенные
    // сложности. Эту проблему можно решить с помощью
    // различных геометрических конструкций, но я
    // воспользуюсь векторами, представляющими двумерные
    // проекции плоскостей, ограничивающих область
    // обзора, а затем - найду перпендикуляры к ним

    // Правая плоскость отсечения; проверьте
    // математические соотношения по схеме, нарисованной
    // на бумаге
    VECTOR3D_INITXYZ(&vn,cam->view_dist,0,-cam->
        viewplane_width/2.0);
    PLANE3D_Init(&cam->rt_clip_plane,&pt_origin,&vn,1);

    // Левая плоскость отсечения. Поскольку правая и
    // левая плоскости отсечения симметричны
    // относительно оси г, достаточно инвертировать
    // координату x вектора нормали к правой плоскости
    // отсечения
    VECTOR3D_INITXYZ(&vn,-cam->view_dist,0,-cam->
        viewplane_width/2.0);
    PLANE3D_Init(&cam->lt_clip_plane,&pt_origin,&vn,1);

    // Верхняя плоскость отсечения; аналогичная
    // конструкция
    VECTOR3D_INITXYZ(&vn,0,cam->view_dist,-cam->
        viewplane_width/2.0);
    PLANE3D_Init(&cam->tp_clip_plane,&pt_origin,&vn,1);

    // Нижняя плоскость отсечения; аналогичное
    // инвертирование
    VECTOR3D_INITXYZ(&vn,0,-cam->view_dist,-cam->
        viewplane_width/2.0);
    PLANE3D_Init(&cam->bt_clip_plane,&pt_origin,&vn,1);
} // else

} // Init_CAM4DV1

```

Пожалуйста, потратьте немного времени и ознакомьтесь с работой этой функции. Она не сложная, но является хорошим примером решения проблем, связанных с камерами и проекциями, а также обработки констант.

Сначала переданные в функцию параметры присваиваются соответствующим переменным, а затем на основе размеров, заданных для поля зрения, и конечных размеров экрана вычисляются все остальные параметры. При этом ось  $x$  используется как базовая. Заметим, что для этой функции не имеет большого значения, какая модель камеры выбрана на данном этапе. Она просто запоминает ее и присваивает значения нескольким полям, но в основном эта вспомогательная функция конструирует матрицу камеры  $tsam$ . А теперь рассмотрим пару небольших примеров.

#### Пример 1

Нужно параметризовать эйлерову камеру нормированными координатами, когда  $FOV=90^\circ$ , расстояние до точки наблюдения равно 1.0, а размеры экрана - 400x400. Функция вычисляет размер поля зрения, который в данном случае составляет 2x2. Описанная модель камеры проиллюстрирована на рис. 7.10.

```
// Инициализация камеры
Init_CAM4DV1(&cam, // Объект, представляющий камеру
CAM_MODEL_EULER, // Модель камеры
&cam_pos, // Начальное положение камеры
&cam_dir, // Начальное направление камеры
NULL // В эйлеровой камере этот параметр не
// используется, задаем значение NULL
50.0, // Ближняя и дальняя плоскости отсечения
500.0,
90.0, // Угол обзора  $\alpha$  в градусах
400, // Конечные размеры экрана
400);
```

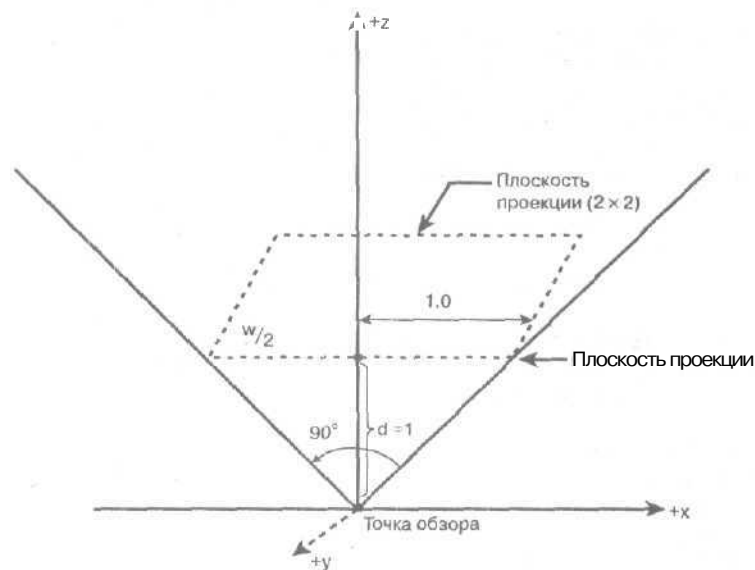


Рис. 7.10. Вычисление нормированных координат камеры для угла обзора, равного  $90^\circ$ , и расстояния до точки наблюдения, равного 1.0

## Пример 2

Нужно параметризовать камеру в модели UVN для упрощенного проецирования на экран, если  $FOV = 90^\circ$ , а размеры экрана составляют 640x480. Функция вычисляет расстояние до точки наблюдения. Описанная модель камеры проиллюстрирована на рис. 7.11.

```
// Инициализация камеры
Init_CAM4DV1(&cam, // Объект, представляющий камеру
CAM_MODEL_UVN, // Модель камеры
&cam_pos, // Начальное положение камеры
&cam_dir, // Начальное направление камеры.
&cam_target, // Начальная целевая точка.
50.0, // Ближняя и дальняя плоскости отсечения
500.0,
90.0, // Угол обзора в градусах
640, // Конечный размер экрана
480);
```

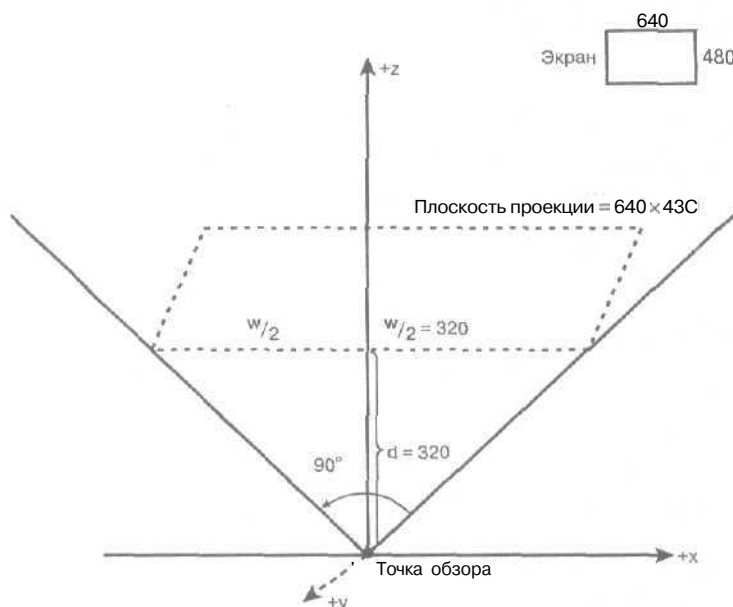


Рис. 7.11. Схема камеры в модели UVN с углом обзора  $90^\circ$  и размерами экрана, равными размерам поля обзора

В обоих рассмотренных случаях размеры поля зрения, фокальное расстояние и отношение ширины экрана к его высоте вычисляются внутри функции. Вообще говоря, рассмотренных в примерах 1 и 2 схем вполне достаточно для всех случаев, с которыми мы столкнемся. Чаще всего мы будем использовать угол обзора  $FOV = 90^\circ$  (изредка  $60^\circ$ ) и нормированную проекцию на поле зрения 2x2 или  $2 \times 2 / \text{aspect\_ratio}$ , если экран не квадратный.

Теперь нам осталось лишь определить способ размещения камеры и вычисления матрицы в объекте камеры. Ниже приведен код функции, при вызове которой зада-

ется положение камеры и углы **Эйлера**, описывающие ее ориентацию. Функция вычисляет все матрицы и другие элементы, которые относятся к камере.

```
void Build_CAM4DV1_Matrix_Euler(CAM4DV1_PTR cam,
                                int cam_rot_seq)
{
    // Эта функция создает матрицу камеры на основе заданных
    // углов Эйлера и сохраняет ее в указанном объекте
    // камеры. Как говорилось в главе 6, для создания
    // матрицы камеры нужно построить матрицу
    // преобразования, имеющую вид:
    // Mcam = mt(-1) * ty(-1) * tx(-1) * mz(-1)
    // т.е. произведение обратных матриц, первая из которых
    // - матрица переноса, а три остальных - матрицы
    // поворотов вокруг осей uxz. Порядок осей, вокруг
    // которых производятся повороты, можно изменять. Это
    // делается с помощью параметра cam_rot_seq, который
    // может принимать значения CAM_ROT_SEQ_XYZ,
    // CAM_ROT_SEQ_YXZ, CAM_ROT_SEQ_ZXY и т.д.

    MATRIX4X4
    mt_inv, // Обратная матрица трансляции.
    tx_inv, // Обратная матрица вращения вокруг оси x
    ty_inv, // Обратная матрица вращения вокруг оси y
    mz_inv, // Обратная матрица вращения вокруг оси z
    mrot, // Произведение обратных матриц вращения
    mtmp; // Временная рабочая матрица

    // Этап 1: создается обратная матрица трансляции.
    // Положение
    Mat_Init_4X4(&mt_inv,
                1, 0, 0, 0,
                0, 1, 0, 0,
                0, 0, 1, 0,
                -cam->pos.x, -cam->pos.y, -cam->pos.z, 1);

    // Этап 2: создается обратная матрица последовательности
    // поворотов. Напомним, что для этого нужно либо
    // транспонировать прямую матрицу поворота, либо
    // изменить знаки углов поворота в прямой матрице

    // Сначала вычислим все три матрицы поворота

    // Извлекаем углы Эйлера
    float theta_x = cam->dir.x;
    float theta_y = cam->dir.y;
    float theta_z = cam->dir.z;

    // Вычисляем синус и косинус
    // от угла поворота вокруг оси x
    float cos_theta = Fast_Cos(theta_x); // cos(-x) - cos(x)
    float sin_theta = -Fast_Sin(theta_x); // sin(-x) - -sin(x)
}
```

```

// Задаем матрицу
Mat_Init_4X4(&mx_inv,
    1,    0,    0, 0,
    0, cos_theta, sin_theta, 0,
    0, -sin_theta, cos_theta, 0,
    0,    0,    0, 1);

// Вычисляем синус и косинус угла поворота вокруг оси y
cos_theta = Fast_Cos(theta_y); // cos(-x) = cos(x)
sin_theta = -Fast_Sin(theta_y); // sin(-x) = -sin(x)

// Задаем матрицу
Mat_Init_4X4(&my_inv,
    cos_theta, 0, -sin_theta, 0,
    0, 1,    0, 0,
    sin_theta, 0, cos_theta, 0,
    0, 0,    0, 1);

// Вычисляем синус и косинус угла поворота вокруг оси z
cos_theta = Fast_Cos(theta_z); // cos(-x) = cos(x)
sin_theta = -Fast_Sin(theta_z); // sin(-x) = -sin(x)

// Задаем матрицу
Mat_Init_4X4(&mz_inv,
    cos_theta, sin_theta, 0, 0,
    -sin_theta, cos_theta, 0, 0,
    0,    0, 1, 0,
    0,    0, 0, 1);

// Вычисляем обратную матрицу последовательных поворотов
switch(cam_rot_seq)
{
    case CAM_ROT_SEQ_XYZ: {
        Mat_Mul_4X4(&mx_inv, &my_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_YXZ: {
        Mat_Mul_4X4(&my_inv, &mx_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_XZY: {
        Mat_Mul_4X4(&mx_inv, &mz_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &my_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_YZX: {
        Mat_Mul_4X4(&my_inv, &mz_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
    } break;
}

```

```

    case CAM_ROT_SEQ_ZYX: {
        Mat_Mul_4X4(&mz_inv, &my_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_ZXY: {
        Mat_Mul_4X4(&mz_inv, &mx_inv, &mtmp);
        Mat_Mul_4X4(&mtmp, &my_inv, &mrot);
    } break;

    default: break;
} // switch

// Теперь матрица mrot равна произведению обратных
// матриц поворота. Умножим ее на обратную матрицу
// переноса - и все готово!
Mat_Mul_4X4(&mt_inv, &mrot, &cam->mcam);

} // Build_CAM4DV1_Matrix_Euler

```

У этой функции два параметра: представляющий камеру объект и флаг, управляющий последовательностью поворотов на величину, заданную углами Эйлера. Сами углы и координаты точки, задающей положение камеры, содержатся в объекте камеры и задаются перед тем, как вызывается функция. Флаги, управляющие последовательностью поворотов, определяются с помощью следующих директив `#define`.

```

// Макроопределения для последовательностей вращения камеры
#define CAM_ROT_SEQ_XYZ 0
#define CAM_ROT_SEQ_YXZ 1
#define CAM_ROT_SEQ_XZY 2
#define CAM_ROT_SEQ_YZX 3
#define CAM_ROT_SEQ_ZYX 4
#define CAM_ROT_SEQ_ZXY 5

```

Предположим, что у нас уже есть объект камеры с именем `cam`, который инициализирован при помощи вызова функции `Init_CAM4DV1()`. Чтобы создать матрицы преобразования камеры для последовательности поворотов `ZYX`, функция `Build_CAM4DV1_Matrix_Euler()` вызывается следующим образом.

```

// Задаем положение камеры
cam.pos.x = 100;
cam.pos.y = 200;
cam.pos.z = 300;

// Задаем углы поворота вокруг осей (в градусах)
cam.dir.x = -45;
cam.dir.y = 0;
cam.dir.z = 0;
// Генерируем матрицу камеры.
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

```

После этого вызова становится доступной матрица `cam.mcam` и появляется возможность перейти с ее помощью из мировых координат в координаты камеры. На рис. 7.12 показаны положение и ориентация виртуальной камеры, заданной с помощью указанных параметров. Однако все еще остаются не определенными матрицы `cam.mpre` и `cam.mscl`.

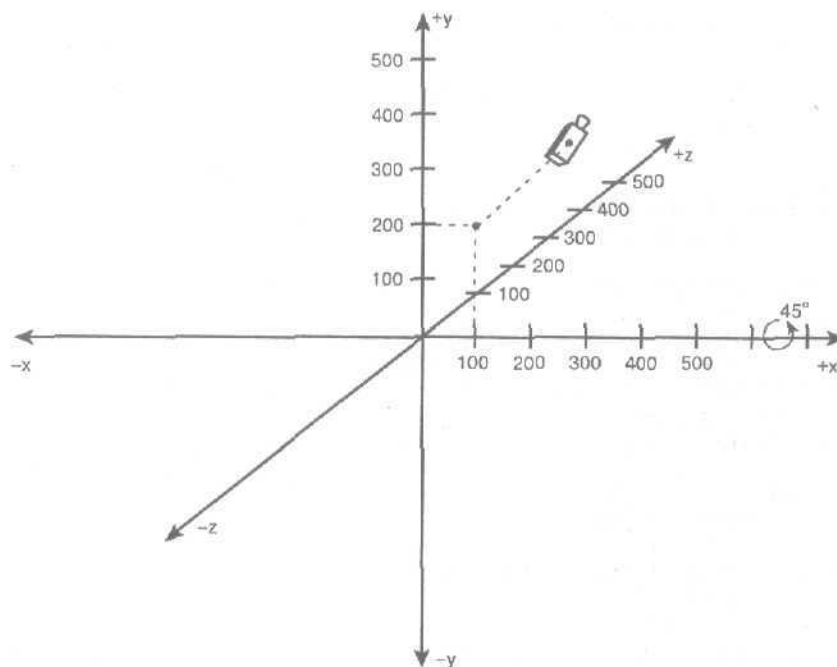


Рис. 7.12. Результат вызова функции, вычисляющей матрицу камеры

Теперь приведем код функции, вычисляющей матрицу камеры в модели UVN.

```
void Build_CAM4DV1_Matrix_UVN(CAM4DV1_PTR cam, int mode)
{
    // Эта функция создает матрицу камеры на основе заданных
    // векторов п (направлен "вперед"), v (направлен
    // "вверх") и и (направлен "вправо" или "влево") и
    // сохраняет ее в указанном объекте камеры. Все
    // необходимые значения извлекаются из самого объекта
    // камеры. Параметр mode позволяет выбирать способ
    // вычисления векторов uvn
    // UVN_MODE_SIMPLE - простой режим низкого уровня, в
    // которой используются целевая и опорная точки
    // UVN_MODE_SPHERICAL - режим использования сферических
    // координат. Компоненты x,y применяются в качестве
    // углов направления и возвышения вектора направления,
    // соответственно, а опорная точка - как обычно, в
    // качестве положения камеры

    MATRIX4X4 mt_inv, // Обратная матрица трансляции
    mt_uvn, // Конечная матрица в модели uvn
    mtmp; // Временная рабочая матрица

    // Этап 1: создается обратная матрица трансляции
    Mat_Init_4X4(&mt_inv,
        1,    0,    0, 0,
        0,    1,    0, 0,
        0,    0,    1, 0,
```

```

-cam->pos.x, -cam->pos.y, -cam->pos.z, 1);

// Этап 2: определяется способ вычисления координат
// целевой точки
if (mode == UVN_MODE_SPHERICAL)
{
    // Используются сферические конструкции
    // Нужно вычислить целевую точку

    // Извлекаем углы возвышения и направления
    float phi = cam->dir.x;
    float theta = cam->dir.y;

    // Вычисляем тригонометрические функции
    float sin_phi = Fast_Sin(phi);
    float cos_phi = Fast_Cos(phi);

    float sin_theta = Fast_Sin(theta);
    float cos_theta = Fast_Cos(theta);

    // Вычисляем координаты x,y,z целевой точки на
    // единичной сфере
    cam->target.x = -1*sin_phi*sin_theta;
    cam->target.y = 1*cos_phi;
    cam->target.z = 1*sin_phi*cos_theta;
} // else

// На данном этапе для вычисления векторов u,v,n
// понадобятся координаты целевой и опорной точек
// Шаг 1: n = <целевое положение - опорное положение>
VECTOR4D_Build(&cam->pos, &cam->target, &cam->n);

// Шаг 2: пусть v = <0,1,0>
VECTOR4D_INITXYZ(&cam->v, 0, 1, 0);

// Шаг 3: u = (v x n)
VECTOR4D_Cross(&cam->v, &cam->n, &cam->u);

// Шаг 4: v = (n x u)
VECTOR4D_Cross(&cam->n, &cam->u, &cam->v);

// Шаг 5: нормируем все векторы
VECTOR4D_Normalize(&cam->u);
VECTOR4D_Normalize(&cam->v);
VECTOR4D_Normalize(&cam->n);

// Создаем матрицу UVN, помещая в ее столбцы координаты
// векторов u,v,n
Mat_Init_4X4(&mt_uvn,
    cam->u.x, cam->v.x, cam->n.x, 0,
    cam->u.y, cam->v.y, cam->n.y, 0,
    cam->u.z, cam->v.z, cam->n.z, 0,
    0, 0, 0, 1);

// Умножаем матрицу трансляции на матрицу uvn и заносим

```

```
// результат в конечную матрицу камеры mcam
Mat_Mul_4X4(&mt_inv, &mt_uvn, &cam->mcam);

} // Build_CAM4DV1_Matrix_UVN
```

Теперь мы можем вернуться к обсуждению нашего трехмерного игрового конвейера.

## Преобразование мировых координат в координаты камеры

Теперь, когда создана модель камеры и появилась возможность задавать ее параметры, можно осуществить преобразование мировых координат в систему координат камеры. Однако сначала разберемся в том, зачем нужны рассмотренные выше функции, моделирующие камеру. И в модели UVN, и в эйлеровой модели задается несколько параметров, описывающих положение камеры и ее ориентацию, после чего конструируется матрица. Этот процесс сводится к двум этапам: переносу и вращению.

На рис. 7.13 проиллюстрировано преобразование мировых координат в координаты камеры. Наблюдение за тем, что происходит в виртуальном трехмерном мире, производится через моделируемую камеру, которая помещается в определенном месте пространства и для которой задается ориентация, а также поле зрения, ограничиваемое по горизонтали и вертикали. Объектив этой виртуальной камеры дает возможность увидеть то, что можно было бы увидеть, если бы камера и мир были реальными. Весь фокус в том, чтобы создать матрицу преобразования, с помощью которой все объекты размещаются таким образом, чтобы они выглядели, как если бы камера находилась в начале координат и была направлена в положительном направлении оси z. При этом упрощаются математические преобразования, необходимые для аксонометрического преобразования на последующих этапах игрового конвейера.

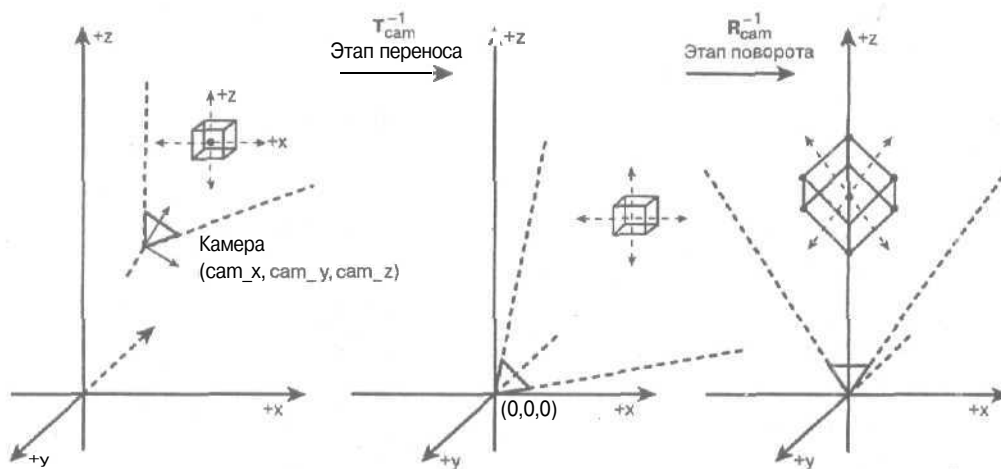


Рис. 7.13. Преобразование мировых координат в координаты камеры

Мы уже знаем, что добиться этого можно следующим путем. Если предположить, что камера расположена в мировой системе координат в точке  $cam\_pos = (cam\_x, cam\_y, cam\_z)$ , определяемой матрицей смещения  $T_{cam}$ , первый этап будет заключаться в переносе всех объектов, вершин, многоугольников и всего остального с помощью обратной матрицы  $T_{cam}^{-1}$ . Величина переноса равна  $-cam\_pos = (-cam\_x, -cam\_y, -cam\_z)$ . Эта матрица выглядит следующим образом:

$$T_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

Преобразование всех вершин с помощью этой матрицы приведет к тому, что все геометрические элементы переместятся относительно камеры, которая находится в начале координат. Следующий шаг, необходимый для построения матрицы преобразования, — поворот. Конечно, он несколько сложнее, и в результате в модели камеры появляется матрица 4x4, определяющая ориентацию камеры. В используемой нами эйлеровой модели камеры ориентацию камеры представляют три угла:  $cam\_ang = (ang\_x, ang\_y, ang\_z)$  (рис. 7.14). Каждый угол соответствует величине поворота вокруг положительного направления каждой оси по часовой стрелке (или против нее в правой системе координат). Эти три преобразования можно разделить на  $R_{camx}$ ,  $R_{camy}$ ,  $R_{camz}$ . Однако, если вы помните, важную роль играет порядок поворотов, т.е. XYZ, YZX или какой-либо другой. Мы решили, что в большинстве случаев будет использоваться порядок YXZ или ZYX.

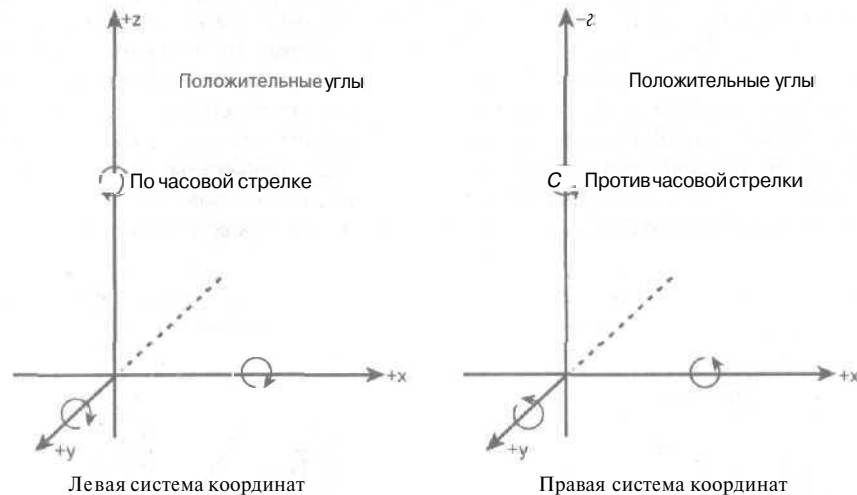


Рис. 7.14. Взаимосвязь углов вращения камеры с координатной системой

Итак, преобразование камеры, соответствующее ее вращению, осуществляется с помощью обратных поворотов в такой последовательности:

$$R_{camy}^{-1} \cdot R_{camx}^{-1} \cdot R_{camz}^{-1},$$

или, возможно, в такой:

$$R_{camz}^{-1} \cdot R_{camy}^{-1} \cdot R_{camx}^{-1}.$$

В итоге получаем, что матрица преобразования камеры  $T_{wc}$  определяется соотношением:

$$T_{wc} = T_{cam}^{-1} \cdot R_{camy}^{-1} \cdot R_{camx}^{-1} \cdot R_{camz}^{-1}.$$

Если же порядок поворотов — ZYX, то это выражение принимает вид:

$$T_{wc} = T_{cam}^{-1} \cdot R_{camz}^{-1} \cdot R_{camy}^{-1} \cdot R_{camx}^{-1}.$$

Суть в том, что сначала выполняется трансляция, а затем — три вращения. Конечно же, эти операции можно объединить в одну, вычислив единую матрицу преобразования  $T_{wc}$  с

помощью представленных выше выражений. Все, что вы должны сделать, — просто преобразовать координаты всех вершин с помощью этой матрицы, которая строится при помощи рассмотренных ранее **функций**. У нас имеется функция, задающая параметры камеры и инициализирующая ее, а также функция `Build_CAM4DV1_Matrix_Euler()`, вычисляющая матрицу перехода из мировых координат в координаты камеры. Кроме того, конечно же, не следует забывать о функции `Build_CAM4DV1_Matrix_UVN()`, которая делает то же самое, но немного по-другому. В ней вместо углов используются концепции векторов, направленных **вверх**, **вправо** и **на целевую точку**. Однако в итоге обе функции находят матрицу преобразования, которую необходимо применить к каждой вершине.

Таким образом, все, что нам нужно, — это функция, в которой к объекту или **списку** визуализации применяется матрица преобразования. Интересно, что у нас уже есть пара функций преобразования общего вида, в которые передается произвольная матрица преобразования и которые способны выполнить преобразование объекта или списка визуализации, — это функции `void Transform_OBJECT4DV1()` и `void Transform_RENDERLIST4DV1()`. Можно было бы воспользоваться любой из них и ни о чем не беспокоиться, но мне больше нравится писать специализированные функции (иногда для каждого этапа игрового конвейера), а не пользоваться функциями общего вида. Это позволит в дальнейшем воспользоваться возможностями оптимизации.

## Преобразование объектов из мировых координат в координаты камеры

Как уже говорилось, у нас есть возможность применить к объекту `OBJECT4DV1` произвольную матрицу преобразования 4x4. Это делается с помощью функции `Transform_OBJECT4DV1()`; рассмотрим сначала этот подход. Предположим, что есть объект `obj`, камера `cam` и что все необходимые структуры и переменные уже **инициализированы**. Поставленная задача решается с помощью следующего кода.

```
// Генерируем матрицу камеры (используется эйлерова камера)
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// Применяем преобразование; убеждаемся, что используются
// мировые координаты, которые преобразуются в координаты
// камеры
Transform_OBJECT4DV1(&obj, &cam.mcam,
    TRANSFORM_TRANS_ONLY, 1);
```

В результате такого приема объект окажется в пространстве камеры, и все будет готово к аксонометрическому проецированию и отображению на экран. С другой стороны, вместо применения функции `Transform_OBJECT4DV1()` можно написать более специализированный код.

```
void World_To_Camera_OBJECT4DV1(CAM4DV1_PTR cam,
    OBJECT4DV1_PTR obj)
{
    // ПРИМЕЧАНИЕ: эта функция основана на применении
    // матрицы. С помощью этой функции выполняется
    // преобразование объекта из мировых координат в
    // координаты камеры на основе заданной матрицы. Однако
    // в ней совсем не учитываются многоугольники; функция
    // работает только с вершинами, содержащимися в списке
    // vlist_trans[]. Это лишь один из способов решения
    // поставленной задачи. Вместо этого можно было бы
```

```

// выполнять преобразование глобального списка
// многоугольников, содержащихся в списке визуализации,
// так как они наверняка представляют геометрию,
// прошедшую этап отбраковки обратных поверхностей (в
// случае его наличия)

// Преобразуем все вершины объекта в координаты камеры.
// Предполагается, что объект уже преобразован в мировые
// координаты, а результат помещен в массив
// vlist_trans[]
for (int vertex = 0; vertex < obj->num_vertices;
    vertex++)
{
    // Преобразуем координаты вершины в систему
    // координат камеры с помощью матрицы mcam. Она
    // должна быть корректной!
    POINT4D presult; // Сохраняем результаты каждого
                    // преобразования

    // Преобразуем координаты точки
    Mat_Mul_VECTOR4D_4X4(&obj->vlist_trans[vertex],
        &cam->mcam, &presult);

    // Сохраняем результаты в том же массиве
    VECTOR4D_COPY(&obj->vlist_trans[vertex], &presult);
} // for vertex

} // World_To_Camera_OBJECT4DV1

```

Эта функция несколько быстрее и компактнее, чем функция Transform\_OBJECT4DV1(). Объект и камера передаются функции как параметры. Ниже приведен пример вызова этой функции при условии, что параметры объекта и камеры уже заданы.

```
World_To_Camera_OBJECT4DV1(&cam, &obj);
```

#### НА ЗАМЕТКУ

В настоящем трехмерном программном игровом процессоре, возможно, стоило бы даже жестко закодировать операции над данными матрицей и вектором. Как показывает опыт, если придерживаться этого правила при написании всех частей процессора, то выигрыш может составить 5-10%. Если это вам нужно — хорошо; в противном случае овчинка не стоит выделки. Лучше заниматься созданием понятных и простых алгоритмов, чем оптимизировать код на таком уровне на стадии разработки.

## Преобразование мировых координат в координаты камеры для списков визуализации

Преобразование мировых координат в координаты камеры для списков визуализации осуществляется точно так же, как и для объектов. Можно воспользоваться общей функцией, осуществляющей преобразование списка визуализации (*rend\_list*). Как и в предыдущем случае, мы рассмотрим оба варианта. В приведенном ниже коде используется уже доступный нам способ.

```

// Генерируем матрицу камеры
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// Применяем преобразование; убеждаемся, что используются
// мировые координаты, которые преобразуются в координаты

```

```
// камеры
Transform_RENDERLIST4DV1(&rend_list, &cam.mcam,
    TRANSFORM_TRANS_ONLY);
```

Обратите внимание, что теперь в списке параметров функции `Transform_RENDERLIST4DV1()` отсутствует флаг `transform_basis`. Причина этого состоит в том, что в списке многоугольников представлено большое количество объектов, и их ориентация не имеет значения, поскольку многоугольники извлечены из объектов и собраны в единый список. Единственной сложностью может стать необходимость обработки каждого многоугольника из списка, тогда как в той версии функции, в которой преобразование объекта `OBJECT4DV1` осуществляется вручную, преобразованию подлежит лишь список вершин. Приведем код соответствующей функции.

```
void World_To_Camera_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. Эта функция основана на применении
    // матрицы. С помощью этой функции выполняется
    // преобразование каждого многоугольника из глобального
    // списка визуализации из мировых координат в координаты
    // камеры с помощью заданной матрицы. Эту функцию
    // следует применять вместо функции, преобразующей
    // объекты, если на предыдущих этапах игрового конвейера
    // разбить каждый объект на отдельные многоугольники и
    // поместить их в глобальный список визуализации.
    // Преобразование объекта в набор многоугольников может
    // осуществляться после отбраковки объектов, перехода в
    // локальные координаты, перехода из локальных координат
    // в мировые и отбраковки обратных поверхностей, чтобы в
    // список вошло минимальное количество многоугольников
    // каждого объекта. В данной функции предполагается,
    // что, по крайней мере, выполнен переход в мировую
    // систему координат и что данные, описывающие
    // многоугольники, находятся в преобразованном массиве

    // Преобразуем параметры каждого многоугольника из
    // списка визуализации в координаты камеры.
    // Предполагается, что данные из списка визуализации уже
    // преобразованы в мировые координаты, и результат
    // помещен в массив tvlist[] каждого объекта,
    // представляющего многоугольник

    for (int poly - 0; poly < rend_list->num_polys; poly++)
    {
        // Запрашиваем параметры каждого многоугольника
        POLYF4DV1_PTR curr_poly -
            rend_list->poly_ptrs[poly];

        // Корректен ли многоугольник?
        // Преобразование многоугольника выполняется только
        // тогда, когда он не отсекается и не
        // отбраковывается, если он активный и видимый.
        // Заметим однако, что концепция "обратных
        // поверхностей" неприменима в игровом процессоре,
```

```

// работающем в каркасном режиме
if ((curr_poly==NULL) ||
    !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // Переходим к следующему
               // многоугольнику

// Все в порядке, осуществляем преобразование
for (int vertex = 0; vertex < 3; vertex++)
{
    // Преобразуем вершину в координаты камеры с
    // помощью матрицы mcam. Она должна быть
    // корректной!
    POINT4D presult; // Сохраняем результаты каждого
                     // преобразования

    // Преобразуем координаты точки
    Mat_Mul_VECTOR4D_4X4(&curr_poly->tvlist[vertex],
        &cam->mcam, &presult);

    // Сохраняем результаты в том же массиве
    VECTOR4D_COPY(&curr_poly->tvlist[vertex],
        &presult);
} // for vertex
} // for poly
} // World_To_Camera_RENDERLIST4DV1

```

Ниже приведен вызов этой функции, с помощью которого выполняется преобразование всего списка визуализации `rend_list`. Как и раньше, предполагается, что представляющий камеру объект `cam` определен и инициализирован.

```
World_To_Camera_RENDERLIST4DV1(&cam, &rend_list);
```

Проще простого! Кстати, замечу, что в данный момент мы находимся примерно на полпути к завершению работы игрового конвейера.

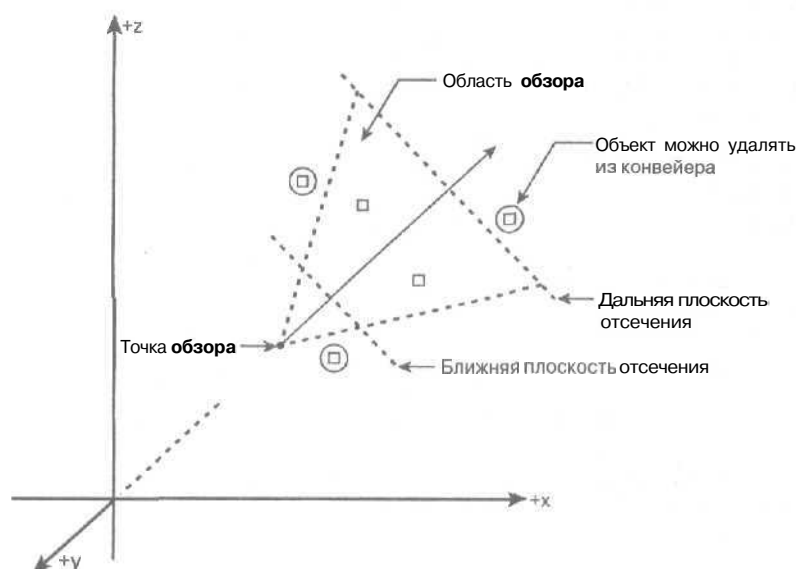
Теперь нам осталось выполнить преобразование в аксонометрические координаты и проецирование на экран. Однако следует сделать некоторые предостережения... Во-первых, еще ничего не было сказано об отсечении, а это очень важная тема. Этот вопрос рассматривался в предыдущей главе, и там было сказано, что отсечение можно выполнять в трехмерном объектном пространстве или в двумерном пространстве изображений. К сожалению, к этому вопросу нам придется вернуться и всерьез над ним поработать.

Однако не можем же мы все время ждать, пока не дойдем до этапа растеризации, ведь нельзя же проецировать вершины многоугольников с отрицательной или, хуже того, нулевой координатой *z* (произойдет ошибка деления на ноль)! Это следует иметь в виду, и с этим что-то нужно делать, но сейчас речь пойдет о том, как удалять из конвейера визуализации целые объекты.

## Отбраковка объектов

*Отбраковка объектов* (object culling) заключается в удалении тех объектов, которые не должны обрабатываться трехмерным игровым конвейером (рис. 7.15). Однако эту операцию можно выполнять как в мировом пространстве, так и в пространстве камеры.

Я предпочитаю делать это в системе координат камеры. Вы можете подумать: "Но если ждать до тех пор, пока не произойдет преобразование в координаты камеры, и лишь затем производить отбраковку, разве не придется напрасно расходовать ресурсы на преобразование объекта в систему координат камеры лишь для того, чтобы понять, что он невидим?" Ответ — это вовсе не обязательно так.



Мировые координаты или координаты камеры

Рис. 7.15. Удаление объектов из трехмерного игрового конвейера

Причина заключается в следующем. Если создать вокруг объекта ограничивающую сферу с центром, совпадающим с центром объекта, и радиусом, равным максимальному радиусу объекта, отбраковку объекта можно будет выполнять путем преобразования из мировых координат в координаты камеры единственной точки — центра тестируемого объекта. После этого проверяется, входит ли ограничивающая сфера в область видимости или выходит за ее пределы. Этот прием подробно описан в предыдущей главе. Если вы помните, задействованные при этом математические операции очень просты и сводятся к проверке того, пересекают ли некоторые лежащие на сфере точки правую, левую, верхнюю и нижнюю плоскости отсечения. Это один из *важнейших* этапов конвейера, лежащего в основе трехмерного игрового процессора, поскольку в большинстве случаев на сцене оказываются лишь несколько объектов. Остальные же находятся *вне* поля зрения или позади наблюдателя, поэтому нет смысла их обрабатывать. Таким образом, в игровой процессор нужно добавлять все описанные функциональные возможности.

Чтобы реализовать отбраковку объектов, нужно создать функцию, в которую передавались бы координаты объекта в данный момент времени, а также его средний или максимальный радиус (*можно* использовать любой из них). Затем в процессе работы функции будут отбраковываться объекты (функция может работать только на уровне объектов, поскольку после разложения на многоугольники и помещения их в основной *список* визуализации такое понятие, как "объект", теряет смысл). Приведем код функции, *осуществляющей* отбраковку объектов.

```
int Cull_OBJECT4DV1(
    OBJECT4DV1_PTRObj, //Объект, который отбраковывается
```

```

CAM4DV1_PTR cam,    // Камера, относительно которой
                    // производится отбраковка
int cull_flags)      // Учитываемые плоскости отсечения
{
    // ПРИМЕЧАНИЕ. 8 функции применяется матрица
    // преобразования. Данная функция отбраковывает объекты,
    // не входящие в область видимости. При этом
    // используются параметры камеры и объекта. С помощью
    // параметра cull_flags определяется, по каким осям
    // следует производить отсечение - по оси x, y, z или по
    // всем, для чего все флаги нужно объединить с помощью
    // побитового оператора ИЛИ. Если объект отбраковывается,
    // его состояние изменяется. В функции предполагается,
    // что и камера, и объект заданы корректно!

    // Этап 1. Преобразуем центр сферы, ограничивающей
    // объект, в систему координат камеры

    POINT4D sphere_pos;    // Сохраняем преобразованные
                          // координаты центра cntr
                          // ограничивающей сферы

    // Преобразуем координаты точки
    Mat_Mul_VECTOR4D_4X4(&obj->world_pos, &cam->mcam,
                          &sphere_pos);

    // Этап 2. Отбраковываем объект на основе заданных
    // флагов
    if (cull_flags & CULL_OBJECT_Z_PLANE)
    {
        // Отбраковываем только по оси z

        // Проверка для дальней плоскости отсечения
        if (((sphere_pos.z - obj->max_radius) >
            cam->far_clip_z) ||
            ((sphere_pos.z + obj->max_radius) <
            cam->near_clip_z))
        {
            SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
            return(1);
        } // if

    } // if

    if (cull_flags & CULL_OBJECT_X_PLANE)
    {
        // Отбраковываем только по оси x
        // Можно было бы воспользоваться уравнением
        // плоскостей, но легче применить свойства подобных
        // треугольников, так как это двумерная задача. Если
        // область видимости имеет форму прямоугольного
        // параллелепипеда, задача становится тривиальной,
        // однако предположим, что это не так
    }
}

```

```

// Выясняем положение самой правой и самой левой
// точек ограничивающей сферы объекта относительно
// правой и левой плоскостей отсечения
float z_test =
    (0.5)*cam->viewplane_width*sphere_pos.z/
    cam->view_dist;

if(((sphere_pos.x-obj->max_radius) > // Правая
    z_test)|| // сторона
    ((sphere_pos.x+obj->max_radius) < // Левая
    -z_test) ) // сторона
{
    SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
    return(1);
} // if
} // if

if (cull_flags & CULL_OBJECT_Y_PLANE)
{
    // Отбраковываем только по оси y
    // Можно было бы воспользоваться уравнением
    // плоскостей, но легче применить свойства подобных
    // треугольников, так как это двумерная задача. Если
    // область видимости имеет форму прямоугольного
    // параллелепипеда, задача становится тривиальной,
    // однако предположим, что это не так

    // Выясняем положение самой верхней и самой нижней
    // точек ограничивающей сферы объекта относительно
    // правой и левой плоскостей отсечения
    float z_test =
        (0.5)*cam->viewplane_height*sphere_pos.z/
        cam->view_dist;

    if(((sphere_pos.y-obj->max_radius) > // Верхняя
        z_test)|| // сторона
        ((sphere_pos.y+obj->max_radius) < // Нижняя
        -z_test)) // сторона
    {
        SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
        return (1);
    } // if
} // if

// Возврат кода, соответствующего отсутствию отбраковки
return(0);

} // Cull_OBJECT4DV1

```

Приведенная выше функция представляет собой простую реализацию описанных в предыдущей главе математических операций, позволяющих выяснить, нужно ли отбра-

ковывать объект, поэтому здесь дополнительных пояснений не требуется. Однако к этой функции можно сделать интересные дополнения. Функция получает объект, для которого нужно определить необходимость отбраковки, текущую камеру (предварительно необходимо вызвать функцию, инициализирующую матрицу камеры) и параметр-флаг `cull_flags`, который определяет набор условных операторов, применяемых для проверки. Рассмотрим его подробнее...

Параметр `cull_flags` управляет тем, по каким осям производится отбраковка. Например, если эта операция выполняется только по оси `z` (т.е. по дальней и ближней плоскостям отсечения), то не имеет значения, находится ли объект в поле зрения по осям `x` и `y`. Однако при этом, по крайней мере, гарантируется, что не нужно будет проецировать многоугольники, которые находятся позади наблюдателя. Вообще говоря, отсечение следует производить по всем шести плоскостям, ограничивающим область видимости, — ведь это так просто и не требует больших затрат ресурсов. Ниже представлены флаги, управляющие отсечением.

```
// Общие флаги отсечения
#define CULL_OBJECT_X_PLANE 0x0001 // Отсечение по оси x
#define CULL_OBJECT_Y_PLANE 0x0002 // Отсечение по оси y
#define CULL_OBJECT_Z_PLANE 0x0004 // Отсечение по оси z
#define CULL_OBJECT_XYZ_PLANES (CULL_OBJECT_X_PLANE | \
                                CULL_OBJECT_Y_PLANE | \
                                CULL_OBJECT_Z_PLANE)
```

Далее их следует просто объединить в логическое выражение с помощью оператора побитового ИЛИ. Например, ниже представлены действия, необходимые для отсечения по всем граням области обзора.

```
if (Cull_OBJECT4DV1(&obj, &cam,
                  CULL_OBJECT_X_PLANE |
                  CULL_OBJECT_Y_PLANE |
                  CULL_OBJECT_Z_PLANE))
    /* Отбраковка объекта */
```

После отбраковки объекта функция изменяет состояние объекта, установив в его флагах бит, указывающий на то, что объект отбракован. Чтобы организовать это, я добавил несколько констант, определяющих состояние объектов класса `OBJECT4DV1`.

```
// Состояния объектов
#define OBJECT4DV1_STATE_ACTIVE 0x0001
#define OBJECT4DV1_STATE_VISIBLE 0x0002
#define OBJECT4DV1_STATE_CULLED 0x0004
```

С помощью присвоения значения флагу `OBJECT4DV1_STATE_CULLED` с последующей его проверкой в функциях визуализации и преобразования можно исключить отбракованные объекты. Кроме того, если объект отбракован в одном кадре, это еще не означает, что он будет отбракован в следующем, и наоборот. Поэтому в каждом игровом кадре нужно сбрасывать значения флагов объекта. Приведем код функции, выполняющей эти действия.

```
void Reset_OBJECT4DV1(OBJECT4DV1_PTR obj)
{
    // Эта функция сбрасывает состояние указанного объекта и
    // готовит его к преобразованиям. Сбрасываются только
    // флаги, соответствующие отбраковке, отсечению и
    // обратным поверхностям, однако здесь можно добавить
    // код, выполняющий подготовку объекта для обработки на
```

```

// последующих этапах игрового процессора

// Объект корректен, разберем его многоугольник за
// многоугольником

// Сбросим флаг объекта, соответствующий отбраковке
RESET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);

// Теперь сделаем то же самое для флагов отсечения и
// обратных поверхностей многоугольников
for (int poly = 0; poly < obj->num_polys; poly++)
{
    // Извлекаем параметры многоугольника
    POLY4DV1_PTR curr_poly = &obj->plist[poly];

    // Сначала выясним, видим ли вообще данный
    // многоугольник
    if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE))
        continue; // Переходим к следующему
                    // многоугольнику

    // Сбросим флаг, соответствующий отсечению и
    // обратной поверхности
    RESET_BIT(curr_poly->state, POLY4DV1_STATE_CLIPPED);
    RESET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

} // for poly

} // Reset_OBJECT4DV1

```

В результате вызова этой функции обнуляются флаги объекта, соответствующие отбраковке, отсечению и обратным поверхностям каждого многоугольника, **входящего** в состав каркаса. Поговорим об обратных поверхностях.

## Удаление обратных поверхностей

Удаление обратных поверхностей проиллюстрировано на рис. 7.16. В ходе этого процесса удаляются все поверхности, нормали к которым направлены *от* наблюдателя. Математические выражения, имеющие отношение к этой операции, описаны в **предыдущей** главе, поэтому я не стану вдаваться здесь в **подробности**. Если итожить суть дела в двух словах, то проверка производится для каждого многоугольника, входящего в состав объекта или списка визуализации. Сначала вычисляются координаты вектора, направленного от многоугольника к точке **наблюдения**, после чего определяется угол между этим вектором и внешней нормалью к многоугольнику. Если величина этого угла превышает  $90^\circ$  (другими словами,  $\mathbf{n} \cdot \mathbf{l} < 0$ ), поверхность многоугольника является обратной и не может быть видимой.

Конечно же, понятие обратной поверхности имеет смысл только для односторонних многоугольников. Для **многоугольников**, видимых с обеих сторон (т.е. **двусторонних**), эта проверка ничего не даст. Обычно данная проверка производится не в системе координат камеры, а в мировом пространстве (поскольку все, что нужно для нее, — это координаты точки наблюдения), тем самым перед переходом из мировых координат в координаты камеры обычно из рассмотрения исключается множество многоугольников. Это весьма подходящий способ не **преобразовывать** в систему координат камеры около 50% геометрических элементов!

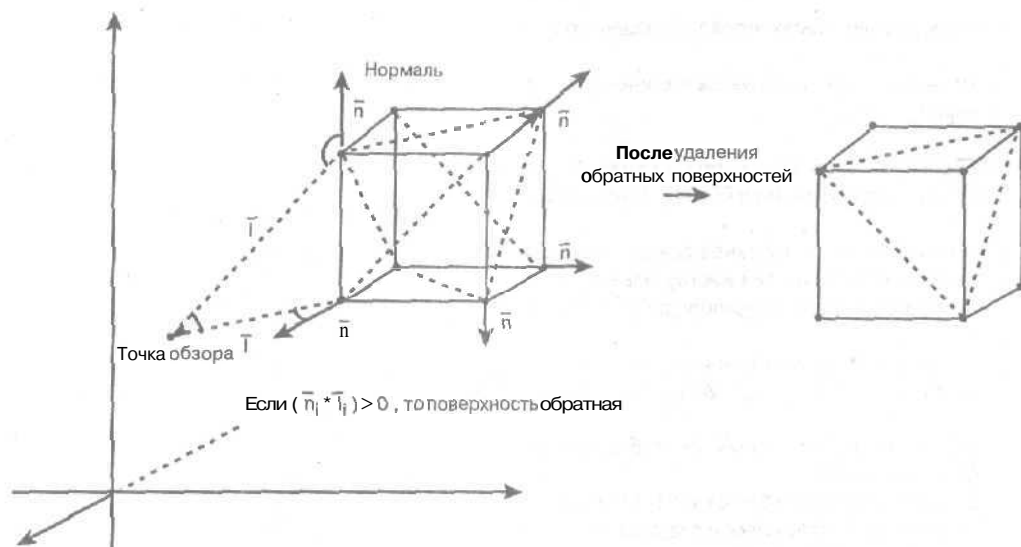


Рис. 7.16. Удаление поверхностей в действии

### Удаление обратных поверхностей в объектах

Приведем код функции, удаляющей обратные поверхности в объектах структуры OBJECT4DV1.

```
void Remove_Backfaces_OBJECT4DV1(OBJECT4DV1_PTR obj,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы,
    // Функция удаляет из каркаса объекта все
    // многоугольники, являющиеся обратными поверхностями,
    // используя при этом параметры вершин, содержащиеся в
    // списке vlist_trans, а также координаты точки, в
    // которой расположена камера. Обратите внимание, что
    // для каждого многоугольника устанавливается только
    // состояние обратной поверхности

    // Проверяем, не отбракован ли объект
    if (obj->state & OBJECT4DV1_STATE_CULLED)
        return;

    // Обрабатываем каждый многоугольник каркаса
    for (int poly=0; poly < obj->num_polys; poly++)
    {
        // Запрашиваем параметры многоугольника
        POLY4DV1_PTR curr_poly = &obj->plist[poly];

        // Корректен ли данный многоугольник?
        // Проверка выполняется только для многоугольников,
        // которые не отбракованы и не отсечены, активны и
        // видимы, а также не являются двусторонними
        if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
```

```

    {curr_poly->state & POLY4DV1_STATE_CLIPPED} ]|
    (curr_poly->attr & POLY4DV1_ATTR_2SIDED) ||
    {curr_poly->state & POLY4DV1_STATE_BACKFACE} )
    continue; // Переходим к следующему
               // многоугольнику

// Извлекаем индексы вершин в основном списке.
// Напомним, что многоугольники НЕ являются
// самодостаточными; в их основе лежит список
// вершин, содержащийся в объекте
int vindex_0 = curr_poly->vert[0];
int vindex_1 = curr_poly->vert[1];
int vindex_2 = curr_poly->vert[2];

// Используется список вершин преобразованного много
// угольника, поскольку удаление обратных
// поверхностей имеет смысл только на этапах игрового
// конвейера, следующих после преобразования в
// мировые координаты

// Необходимо вычислить нормаль к поверхности
// данного многоугольника. Напомним, что обход
// вершин осуществляется по часовой стрелке, так что
//  $u = p0 \rightarrow p1, v = p0 \rightarrow p2, n = u \times v$ 
VECTOR4D u, v, n;

// Вычисляем координаты векторов u, v
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
               &obj->vlist_trans[ vindex_1 ], &u);
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
               &obj->vlist_trans[ vindex_2 ], &v);

// Находим их векторное произведение
VECTOR4D_Cross(&u, &v, &n);

// Создаем вектор, направленный на точку наблюдения
VECTOR4D view;
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
               &cam->pos, &view);

// И наконец, вычисляем скалярное произведение
float dp = VECTOR4D_Dot(&n, &view);

// Если оно > 0, то поверхность считается видимой,
// если = 0 - это злая шутка, а если < 0, то
// поверхность невидима
if (dp <= 0.0 )
    SET_BIT(curr_poly->state,
            POLY4DV1_STATE_BACKFACE);

} //for poly
} // Remove_Backfaces_OBJECT4DV1

```

Как видите, весьма целесообразным оказалось то, что ранее мы написали все математические функции для работы с матрицами и векторами. Это облегчило разработку при-

веденной выше функции. Суть вот в чем: функция получает объект, преобразует его в список многоугольников и проверяет, все ли многоугольники являются активными, видимыми и односторонними. После этого проверяется, не является ли данная поверхность обратной по отношению к *текущей* точке наблюдения, координаты которой *определяются* объектом, представляющем камеру. Заметим, что для этой функции не нужно задавать матрицу *камеры* `mcam`, поскольку для ее работы достаточно располагать сведениями о положении камеры. Ниже представлен пример *использования* данной функции.

```
Remove_Backfaces_OBJECT4DV1(&obj,&cam);
```

Трудно? Ни капельки! Функция изменяет флаги всех *входящих* в список многоугольников, являющихся обратными поверхностями. Точнее говоря, в каждом таком многоугольнике устанавливается бит обратной поверхности. Это делается следующим образом.

```
SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);
```

СОВЕТ

Убедитесь, что удаление обратных поверхностей происходит *после* удаления объектов и *перед* переходом из мировых координат в координаты камеры.

## Удаление обратных поверхностей в списках визуализации

Если перед удалением обратных поверхностей было решено собрать информацию обо всех объектах в одном месте, и теперь у нас нет отдельных объектов, а есть только общий список *многоугольников*, то в процессе удаления обратных поверхностей нам придется работать с этим списком (конечно же, если этот шаг был выполнен заранее, когда в нашем распоряжении еще были объекты, то говорить просто не о чем). Функция, которая проверяет наличие обратных поверхностей в объекте `RENDERLIST4DV1`, почти идентична той, которая выполняет аналогичную проверку для объекта `OBJECT4DV1`, — просто в ней немного по-другому организован доступ к структурам данных. Ниже приведен код этой функции.

```
void Remove_Backfaces_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
    // Функция удаляет из списка многоугольников обратные
    // поверхности. Это удаление выполняется на основе
    // данных, содержащихся в списке многоугольников, и
    // координат точки, в которой находится камера. Обратите
    // внимание, что в функции для каждого многоугольника
    // устанавливается только состояние обратной поверхности

    for (int poly = 0; poly < rend_list->num_polys; poly++)
    {
        // Запрашиваем параметры текущего многоугольника
        POLYF4DV1_PTR curr_poly=rend_list->poly_ptrs[poly];

        // Корректен ли данный многоугольник?
        // Проверка выполняется только для многоугольников,
        // которые не отбракованы и не отсечены, активны и
        // видимы, а также не являются двусторонними
        if ((curr_poly==NULL) ||
            !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
            (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
            (curr_poly->attr & POLY4DV1_ATTR_2SIDED) ||
```

```

(curr_poly->state & POLY4DV1_STATE_BACKFACE) )
continue;//Переходкследующемумногоугольнику

// Нужно вычислить нормаль к поверхности данного
// многоугольника. Напомним, что обход вершин
// осуществляется по часовой стрелке, так что
// u = p0->p1, v=p0->p2, n=uxv
VECTOR4D u, v, n;

// Вычисляем координаты векторов u, v
VECTOR4D_Build(&curr_poly->tvlist[0],
               &curr_poly->tvlist[1], &u);
VECTOR4D_Build(&curr_poly->tvlist[0],
               &curr_poly->tvlist[2], &v);

// Находим их векторное произведение
VECTOR4D_Cross(&u, &v, &n);

// Создаем вектор, направленный на точку наблюдения
VECTOR4D view;
VECTOR4D_Build(&curr_poly->tvlist[0], &cam->pos,
               &view);

// И наконец, вычисляем скалярное произведение
float dp = VECTOR4D_Dot(&n, &view);

// Если произведение > 0, то поверхность видимая,
// если * 0, это злая шутка, а если < 0, то
// поверхность невидима
if (dp <= 0.0)
    SET_BIT(curr_poly->state,
            POLY4DV1_STATE_BACKFACE);

} // for poly

} // Remove_Backfaces_RENDERLIST4DV1

```

Так же, как и в версии функции, работающей с объектами класса **OBJECT4DV1**, обратные поверхности следует удалять перед переходом из мировых координат в **координаты** камеры, поскольку в ходе этого преобразования напрасно расходовались бы ресурсы для обработки **многоугольников**, **обращенных** к наблюдателю обратной стороной! Ниже приведен пример вызова функции.

```
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);
```

После этого во всех многоугольниках, представляющих собой обратные поверхности, будет установлен бит **POLY4DV1\_STATE\_BACKFACE**.

## АксонOMETрическое преобразование

Следующий этап трехмерного игрового конвейера — выполнение аксонометрического преобразования. Данный процесс показан на рис. 7.17, где еще раз приведены описывающие его формулы. Если имеется точка  $p(x_{world}, y_{world}, z_{world}, 1)$ , то координаты ее аксонометрической проекции определяются соотношениями:

$$x_{\text{per}} = d \cdot x_{\text{world}} / z_{\text{world}},$$

$$y_{\text{per}} = d \cdot \text{ar} \cdot y_{\text{world}} / z_{\text{world}}.$$

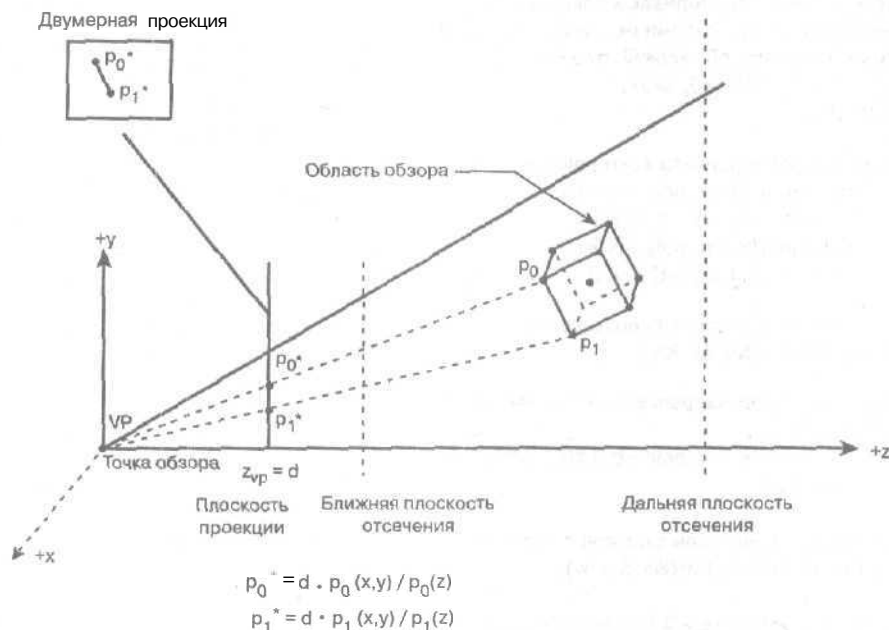


Рис. 7.17. Схема аксонометрического преобразования

Напомним, что если переменной, в которой хранится расстояние до точки наблюдения (обозначенной как  $d$ ), присвоить значение 1.0, то координаты, описывающие положение точки на плоскости наблюдения, будут нормированными и будут изменяться в следующих диапазонах:

по оси  $x$ :  $x \in [-1, 1]$ ,

по оси  $y$ :  $y \in [-1/\text{ar}, 1/\text{ar}]$ ,

где  $\text{ar}$  — это форматное отношение (отношение ширины экрана к его высоте).

Кроме того, если расстояние до точки наблюдения равно 1.0 и область обзора имеет форму квадрата, то и по горизонтали, и по вертикали образуется угол обзора, равный  $90^\circ$ . Далее, если мы хотим иметь большую свободу в выборе поля зрения или расстояния до точки наблюдения и получить возможность выполнять проецирование на область наблюдения произвольных размеров, то следует учитывать, что расстояние до точки наблюдения и размеры области обзора связаны между собой следующими соотношениями, играющими важную роль в аксонометрических преобразованиях:

$$d = \text{width} \cdot \text{tg}(\theta_h/2)/2,$$

$$x_{\text{per}} = x \cdot x_{\text{world}} / z_{\text{world}},$$

$$y_{\text{per}} = d \cdot y_{\text{world}} \cdot \text{ar} / z_{\text{world}}.$$

Ну все, достаточно всех этих обзоров. Наша основная цель — реализовать аксонометрическое преобразование в виде кода. Здесь возможны два различных подхода: 1) выполнить математические операции "вручную", 2) выполнить это преобразование с помощью матриц. Однако со вторым методом связана одна проблема. Как вы помните, с помощью

матриц невозможно реализовать деление на  $z$ , и поэтому нам необходимо воспользоваться четырехмерными однородными координатами, а затем преобразовать результат, полученный с помощью действий с матрицами, назад в трехмерные неоднородные координаты. Матричное умножение само по себе является **нерациональным**, а поскольку при его использовании для аксонометрического преобразования требуются дополнительные действия для обратного перехода к неоднородным координатам, применение матричных операций для аксонометрического преобразования — далеко не самый оптимальный способ с точки зрения программного обеспечения. Тем не менее, еще раз отметим, что многие аппаратные реализации предпочитают иметь дело с **матрицами**. Поэтому мы рассмотрим применение обоих методов к объектам классов `OBJECT4DV1` и `RENDERLIST4DV1`.

### Аксонометрическое преобразование объектов

Объект камеры обладает всей информацией, необходимой для выполнения аксонометрических преобразований, включая расстояние до точки наблюдения, ширину и высоту области обзора, а также величину поля обзора, так что соответствующей функции достаточно передать только объект `OBJECT4DV1` и камеру.

```
void Camera_To_Perspective_OBJECT4DV1(OBJECT4DV1_PTR obj,
                                       CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
    // Функция выполняет аксонометрическое преобразование
    // объекта на основе информации, содержащейся в объекте
    // камеры. Функция не имеет дела с многоугольниками; она
    // работает со списком вершин, хранящемся в массиве
    // vlist_trans[]

    // Выполняем аксонометрическое преобразование всех
    // вершин объекта. Предполагается, что объект уже
    // преобразован в координаты камеры, а результаты
    // находятся в массиве vlist_trans[]
    for (int vertex = 0; vertex < obj->num_vertices;
         vertex++)
    {
        float2 = obj->vlist_trans[vertex].z;

        // Преобразуем вершину с помощью параметров камеры
        obj->vlist_trans[vertex].x = cam->view_dist_h *
            obj->vlist_trans[vertex].x/z;
        obj->vlist_trans[vertex].y = cam->view_dist_v *
            obj->vlist_trans[vertex].y *
            cam->aspect_ratio/z;
        // z = z

        // Заметим, что деление на однородную координату w
        // не выполняется, поскольку в этой версии функции
        // операции над матрицами не используются

    } // for vertex
} // Camera_To_Perspective_OBJECT4DV1
```

Как видите, это очень короткая функция. По сути, в ней координаты каждой вершины умножаются на расстояние до точки наблюдения (по осям  $x$  и  $y$ , что с технической точки зрения одно и то же, хотя в случае с неквадратной областью обзора следует не забывать о форматном соотношении), а затем делятся на величину  $z$ . Пример вызова функции представлен ниже,

```
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);
```

НА ЗАМЕТКУ

В этой функции не производится проверка на отрицательные или нулевые значения  $z$ . От таких вершин нужно избавиться на предыдущей стадии отбраковки или отсеечения. А пока мы считаем, что преобразованию подлежат все вершины.

А что, если **проецирование** нужно реализовать с помощью матрицы? Никаких проблем. Для этого можно воспользоваться функцией общего назначения `Transform_OBJECT4DV1()`, передавая ей соответствующую матрицу преобразования, которая создается с помощью функции

```
void Build_Camera_To_Perspective_MATRIX4X4(CAM4DV1_PTR cam,
      MATRIX4X4_PTR m)
```

```
{
// В этой функции создается матрица аксонометрического
// преобразования. В большинстве случаев камера будет
// обладать нормированной плоскостью наблюдения 2x2 и
// углом обзора, равным 90°. Необходимость этой
// функции продиктована тем, что в дальнейшем нам
// понадобится матрица перехода из аксонометрических
// координат в экранные, выполняющая масштабное
// преобразование нормированных координат.
// Предполагается, что мы работаем с четырехмерными
// однородными координатами и на некотором этапе
// последует переход из четырехмерного пространства в
// трехмерное. Этот переход можно выполнить как сразу
// после аксонометрического преобразования вершин, так и
// после преобразования в экранные координаты
```

```
Mat_Init_4X4(m,
  cam->view_dist_h, 0, 0, 0,
  0, cam->view_dist_v*,
  cam->aspect_ratio, 0, 0,
  0, 0, 1, 1,
  0, 0, 0, 0);
```

```
} // Build_Camera_To_Perspective_MATRIX4X4
```

Далее выполняется обычное преобразование объекта `OBJECT4DV1` с помощью упоминавшейся функции `Transform_OBJECT4DV1()`.

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);
Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY, 1);
```

Конечно же, есть еще одна деталь. После этого преобразования координата  $w$  каждой вершины  $[x \ y \ z \ w]$  уже не равна 1.0, поэтому теперь полученные однородные координаты перед использованием следует преобразовать обратно в неоднородные, для чего все компоненты нужно поделить на компоненту  $w$ . Приведем код функции, выполняющей эту операцию над объектами `OBJECT4DV1`.

```

void Convert_From_Homogeneous4D_OBJECT4DV1(
    OBJECT4DV1_PTR obj)
{
    // Эта функция преобразует все вершины в преобразованном
    // списке вершин из четырехмерных однородных координат в
    // трехмерные неоднородные. Для этого компоненты x,y,z
    // делятся на компоненту w

    for (int vertex = 0; vertex < obj->num_vertices;
        vertex++)
    {
        // Переход к неоднородным координатам
        VECTOR4D_DIV_BY_W(&obj->vlist_trans[vertex]);
    } // for vertex

} // Convert_From_Homogeneous4D_OBJECT4DV1

```

Таким образом, в качестве последнего шага для получения реальных трехмерных координат следует вызвать эту функцию.

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

### Аксонометрическое преобразование списков визуализации

Аксонометрическое преобразование объектов типа `RENDERLIST4DV1` выполняется аналогично преобразованию объектов типа `OBJECT4DV1`. Предназначенные для этого функции более-менее одинаковы, однако доступ к данным в них организован по-разному. Приведем код предварительной версии функции, предназначенной для аксонометрического преобразования без применения матриц.

```

void Camera_To_Perspective_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В данной функции не используются матрицы.
    // Эта функция выполняет аксонометрическое
    // преобразование всех многоугольников, входящих в
    // глобальный список визуализации, с использованием
    // параметров камеры. Данную функцию следует вызывать
    // вместо аналогичной функции, выполняющей
    // преобразование объекта, если на предшествующих
    // стадиях игрового конвейера каждый объект
    // трансформирован в список многоугольников, которые
    // затем добавлены в глобальный список визуализации

    // Выполняем аксонометрическое преобразование всех
    // многоугольников из списка визуализации
    for (int poly = 0; poly < rend_list->num_polys; poly++)
    {
        // Запрашиваем параметры текущего многоугольника
        POLYF4DV1_PTR curr_poly =
            rend_list->poly_ptrs[poly];

        // Является ли данный многоугольник корректным?
        // Многоугольник подлежит преобразованию, только если

```

```

// он не отсечен, не отбракован, если он активный и
// видимый. Однако заметим, что концепция обратных
// поверхностей неприменима в игровом процессоре,
// работающем в каркасном режиме
if ((curr_poly==NULL) ||
    !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; //Переход к следующему многоугольнику

// Все в порядке, выполняем преобразование
for (int vertex * 0; vertex < 3; vertex++)
{
    float z = curr_poly->tvlist[vertex].z;

    // Преобразуем вершину с помощью параметров
    // камеры
    curr_poly->tvlist[vertex].x = cam->view_dist_h *
        curr_poly->tvlist[vertex].x/z;
    curr_poly->tvlist[vertex].y = cam->view_dist_v *
        curr_poly->tvlist[vertex].y *
        cam->aspect_ratio/z;
    //z-z

    //Заметим, что деление на однородную координату
    // w не выполняется, поскольку в этой версии
    // функции не используются операции с матрицами

} //for vertex

} // for poly

} // Camera_To_Perspective_RENDERLIST4DV1

Эта функция применяется точно так же, как и функция для работы с объектами
OBJECT4DV1.

Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

Для реализации матричной версии данной функции сначала путем вызова функции
Build_Camera_To_Perspective_MATRIX4X4() создается соответствующая матрица, а затем вы-
зывается функция Transform_RENDERLIST4DV1().

MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_RENDERLIST4DV1(&rend_list, &mper,
    TRANSFORM_TRANS_ONLY);

Что делать дальше — догадайтесь сами. Правильно — нужно преобразовать однородные
координаты в неоднородные, потому что компонента w теперь не равна 1.0 (фактически
w = z ). Поэтому нужно вызвать функцию, выполняющую это преобразование.

void Convert_From_Homogeneous4D_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list)
(

```

```

// Эта функция преобразует вершины всех корректных
// многоугольников из четырехмерных однородных координат
// в обычные трехмерные путем деления каждой компоненты
// x,y,z на компоненту w

for (int poly = 0; poly < rend_list->num_polys; poly++)
{
    // Запрос параметров текущего многоугольника
    POLY4DV1_PTR curr_poly=rend_list->poly_ptrs[poly];

    // Является ли данный многоугольник корректным?
    // Многоугольник подлежит преобразованию, только если
    // он не отсечен, не отбракован, если он активный и
    // видимый. Однако заметим, что концепция обратных
    // поверхностей неприменима в игровом процессоре,
    // работающем в каркасном режиме
    if ((curr_poly==NULL) ||
        !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
        (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
        (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
        continue; // Переход к следующему многоугольнику

    // Все в порядке, выполняем преобразование
    for (int vertex = 0; vertex < 3; vertex++)
    {
        // Переходим к неоднородным координатам
        VECTOR4D_DIV_BY_W(&curr_poly->tvlist[vertex]);
    } // for vertex

} // for poly

} // Convert_From_Homogeneous4D_RENDERLIST4DV1

```

В результате последовательность вызовов будет выглядеть таким образом.

```

MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);
Transform_RENDERLIST4DV1(&rend_list, &mper,
    TRANSFORM_TRANS_ONLY);
Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);

```

Теперь виртуальное изображение помещено в область *обзора*. Его нужно спроецировать на экран, и это будет последний этап в последовательности преобразований (конечно же, за исключением растеризации).

## Преобразование аксонометрических координат в экранные

Преобразование аксонометрических координат в экранные — одна из последних стадий трехмерного игрового процессора. На вход функции, выполняющей эту операцию, должны передаваться координаты области обзора, над которыми будет выполняться масштабирование для получения экранных координат (рис. 7.18). В ходе преобразования необходимо учитывать, что начало координат большинства экранов находится в левом

верхнем углу, а ось  $y$  направлена в противоположном направлении относительно одноименной оси в обычной декартовой системе координат. Конечно, если в процессе аксонометрического преобразования размеры поля обзора выбраны такими же, как и размеры экрана, масштабирование не понадобится. Однако в большинстве случаев сдвиг начала координат и инвертирование оси  $y$  остается актуальным, потому что в процессе проецирования предполагается, что центр поля обзора находится в начале координат, ось  $x$  направлена **вправо**, а ось  $y$  — **вверх**. Растровые экраны, как уже было сказано, имеют систему координат с центром в левом верхнем углу и направленной вниз осью  $y$ . Поэтому определенных преобразований, связанных с переходом в экранные координаты, избежать никак не удастся.

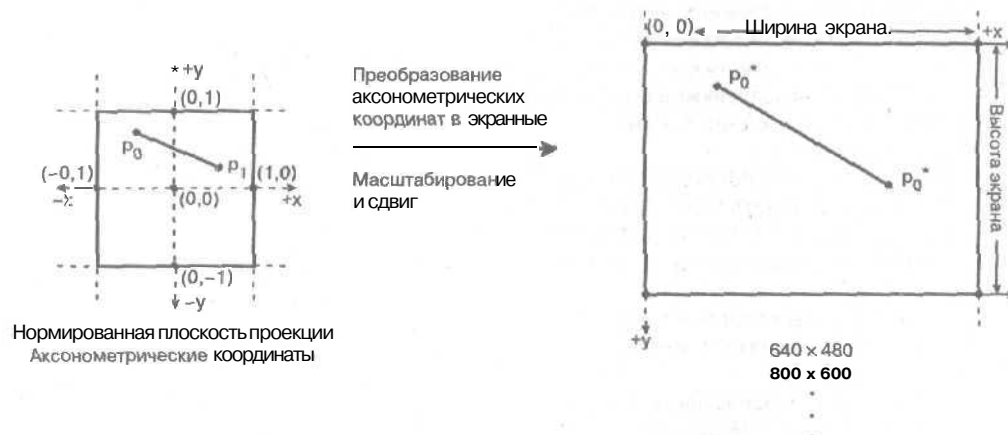


Рис. 7.18. Преобразование аксонометрических координат в экранные

Учитывая все сказанное выше, рассмотрим несколько способов перехода к экранным координатам как для объектов, так и для списков **визуализации**. При этом следует иметь в виду, что в ходе этих преобразований предполагается, что аксонометрические координаты были нормированы (другими словами, проецирование **осуществлялось** на квадратное поле обзора  $2 \times 2$  или отличное от квадратного поля обзора  $2 \times 2/\text{ar}$ ), а потому необходимо масштабирование. В следующем разделе мы увидим, каким образом аксонометрическое преобразование и преобразование в экранные координаты можно выполнить как один этап.

### Преобразование объектов в экранные координаты

Сначала рассмотрим функцию, с помощью которой можно вручную выполнить преобразование аксонометрических координат в экранные. Эта функция предполагает, что аксонометрические координаты нормированы, поэтому необходимо **осуществить** их масштабирование, сдвиг и инвертирование оси  $y$  (все это описано в главе 6, “Введение в трехмерную графику”). Приведем листинг этой функции.

```
void Perspective_To_Screen_OBJECT4DV1(OBJECT4DV1_PTR obj,
                                       CAM4DV1_PTR cam)
```

```
!
```

```
// ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
```

```
// Функция преобразует аксонометрические координаты
```

```
// объекта в экранные с использованием указанных
```

```

// параметров экрана. В функции не рассматриваются
// многоугольники, она работает только с вершинами,
// содержащимися в массиве vlist_trans[]. Заметим, что
// функция разработана исключительно в экспериментальных
// целях. Маловероятно, чтобы объекты оставались
// цельными на таких поздних этапах игрового конвейера.
// Данная функция должна вызываться после
// аксонометрического преобразования объекта

// Преобразуем все вершины объекта в экранные
// координаты. Предполагается, что преобразование в
// аксонометрические координаты уже выполнено, а
// результат помещен в массив vlist_trans[]

float alpha = (0.5*cam->viewport_width - 0.5);
float beta = (0.5*cam->viewport_height - 0.5);

for (int vertex = 0; vertex < obj->num_vertices;
     vertex++)
{
    // Предполагается, что вершины определены в
    // нормированных координатах в интервале от -1 до 1
    // по каждой оси. Выполняем их масштабирование в
    // соответствии с размерами экрана, а также
    // инвертируем ось y и выполняем проецирование на
    // экран

    // Преобразуем вершину с использованием параметров
    // экрана
    obj->vlist_trans[vertex].x = alpha + alpha*obj->
        vlist_trans[vertex].x;
    obj->vlist_trans[vertex].y = beta - beta *obj->
        vlist_trans[vertex].y;
} // for vertex

} // Perspective_To_Screen_OBJECT4DV1

```

Эта функция отображает координаты  $x \in [-1, 1]$  и  $y \in [-1, 1]$  (нормированные координаты поля обзора) на растровый экран. Однако здесь следует учесть одну деталь: переход к однородным четырехмерным координатам не является обязательным, потому что аксонометрическое преобразование (при котором требовалось деление на  $z$ ) уже выполнено. После преобразования в экранные координаты компоненты  $(x, y)$  каждой вершины представляют ее реальное положение на растровом экране и готовы к обработке в последующих этапах.

Может случиться так, что в процессе аксонометрического преобразования деление на  $z$  не производилось, т.е. это преобразование осуществлялось с помощью матриц, но полученные координаты остались в четырехмерной однородной форме (когда компонента  $w$  не равна 1.0). В этом случае данную функцию также можно использовать, но на определенном этапе нужно будет разделить координаты каждой вершины на компоненту  $w$ , осуществив тем самым преобразование однородных координат в неоднородные. Однако в большинстве случаев координаты не остаются четырехмерными на всех этапах вплоть

до перехода в экранную систему координат; тем не менее, нет правила, запрещающего это. В любом случае, использование данной функции выглядит следующим образом.

```
// Сначала переходим из координат камеры к аксонометрическим
// координатам. После этого координаты становятся
// неоднородны
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);
```

```
// Далее переходим к экранным координатам
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);
```

Все легко — даже слишком легко!

Теперь поговорим о том, как выполнить преобразование аксонометрических координат в координаты камеры с помощью матриц. Загвоздка вот в чем; применяющиеся матрицы могут различаться в зависимости от того, заданы ли аксонометрические координаты в четырехмерной однородной форме или в них уже выполнено деление на компоненту  $w$ . В таком случае мы имеем дело с обычными трехмерными координатами (по сути даже двумерными, потому что нас интересуют только компоненты  $x$  и  $y$ ). Мы рассмотрим обе версии данной функции — и ту, которая работает с четырехмерными координатами, и ту, которая работает с трехмерными.

Предположим, что аксонометрическое преобразование выполнено с помощью следующего кода.

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);
Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY, 1);
```

Тогда полученные результаты будут представлены в четырехмерном виде, и чтобы получить конечные значения координат каждой вершины, их нужно будет разделить на величину  $w$ . Однако можно обойтись и без этого и сразу же приступить к переходу из аксонометрических координат в экранные в четырехмерном пространстве. Для этого нужно сконструировать матрицу с помощью следующей функции.

```
void Build_Perspective_To_Screen_4D_MATRIX4X4(
    CAM4DV1_PTR cam,
    MATRIX4X4_PTR m)
{
    // Эта функция генерирует матрицу перехода из
    // аксонометрических координат в экранные.
    // Предполагается, что переход будет выполнен в
    // однородных координатах, и во время растеризации будет
    // произведено преобразование четырехмерных однородных
    // координат в трехмерные. Кроме того, в процессе
    // визуализации во внимание принимаются только
    // координаты  $x$  и  $y$ . Предполагается, что
    // аксонометрические координаты нормированы, и поле
    // обзора имеет размеры  $2 \times 2$ , т.е. координата  $x$  меняется
    // в диапазоне от -1 до 1, а координата  $y$  - от  $-1/\alpha$  до
    //  $1/\alpha$ 

    float alpha = (0.5*cam->viewport_width - 0.5);
    float beta = (0.5*cam->viewport_height - 0.5);

    Mat_Init_4X4(m, alpha, 0, 0, 0,
                 0, -beta, 0, 0,
```

```
alpha, beta, 1, 0,
0, 0, 0, 1);
```

```
} // Build_Perspective_To_Screen_4D_MATRIX4X4()
```

Теперь можно вызвать эту функцию, чтобы построить необходимую матрицу, а затем преобразовать объект. Это делается следующим образом.

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D_MATRIX4X4(&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mser, TRANSFORM_TRANS_ONLY, 1);
```

У нас уже почти получены экранные координаты, осталось только перевести их в неоднородную форму, воспользовавшись вызовом **функции**

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

**СОВЕТ**

Это может показаться странным, однако **всегда** нужно помнить о том, что НЕВОЗМОЖНО выполнить деление на  $z$  с помощью матричных операций, поэтому компоненту  $z$  следует преобразовать в компоненту  $w$  и выполнить деление на нее. Только таким образом мы можем выполнить аксонометрическое (неафинное) преобразование при помощи матричных операций. В этом и заключается проблема, связанная с использованием четырехмерных координат. Работать с ними удобно, так как они позволяют пользоваться матрицами  $4 \times 4$ . Однако в большинстве случаев я предпочитаю "насильно" соблюдать равенство  $w = 1$  на протяжении всех этапов и осуществлять аксонометрическое преобразование вручную, самостоятельно организуя деление на  $z$  вместо того, чтобы специально выделять этап перехода от однородных координат в неоднородные, когда все делится на компоненту  $w$ .

Итак, это был трудный способ. А теперь предположим, что вершины, из которых состоит объект, заданы не в **четырёхмерном**, а в трёхмерном пространстве, и готовы к дальнейшей обработке. Все, что нужно в этом **случае**, — матричная версия функции перехода к экранным координатам, осуществляющая те же математические операции, что и функция `Perspective_To_Screen_OBJECT4DV1()`. Таким образом, в этой новой функции предполагается, что координаты вершин трёхмерные, а компонента  $w = 1$ . Приведем код **функции**, конструирующей матрицу перехода.

```
void Build_Perspective_To_Screen_MATRIX4X4(CAM4DV1_PTR cam,
MATRIX4X4_PTR m)
```

```
{
// Эта функция создает матрицу перехода из
// аксонометрических координат в экранные.
// Предполагается, что этот переход осуществляется в
// двумерном/трехмерном пространстве, т.е. что
// аксонометрические координаты перед применением
// матрицы уже преобразованы из четырехмерных в
// трехмерные. Задача данной матрицы -осуществить
// масштабирование и сдвиг аксонометрических координат,
// чтобы перейти к экранным координатам. Поэтому она
// строится в предположении, что аксонометрические
// координаты заданы в нормированном виде для поля
// обзора  $2x2/af$ , т.е.  $x$  изменяется от -1 до 1, а  $y$  от
//  $-1/af$  до  $1/af$ . Единственное отличие данной функции от
// той версии, в которой предполагается, что координаты
// однородны, заключается в том, что в последнем столбце
// нет присвоения  $w=z$ . Компоненты  $z$  и  $w$  фактически не
```

```

// участвуют в вычислениях, так как, согласно
// предположению, перед применением этой матрицы
// координаты уже преобразованы из четырехмерных в
// трехмерные

float alpha = (0.5*cam->viewport_width - 0.5);
float beta = (0.5*cam->viewport_height - 0.5);

Mat_Init_4X4(m, alpha, 0, 0, 0,
             0, -beta, 0, 0,
             alpha, beta, 1, 0,
             0, 0, 0, 1);

} // Build_Perspective_To_Screen_MATRIX4X4

```

Здесь предполагается, что координаты вершин уже заданы в трехмерном виде (фактически в двумерном, к тому же в нормированных аксонометрических координатах). Чтобы воспользоваться функцией, нужно поступить **следующим** образом.

```

MATRIX4X4 mscr;
Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mscr, TRANSFORM_TRANS_ONLY, 1);

```

Все эти примеры были **приведены**, чтобы **продемонстрировать**, что при переходе из четырехмерного пространства в трехмерное (обычно это происходит в ходе аксонометрического преобразования) математические преобразования, выполняемые на определенных этапах игрового конвейера, а также использующиеся при этом матрицы слегка изменяются. В результате приходится иметь дело с одним из двух возможных вариантов. Лично я предпочитаю выполнять преобразование из четырехмерного пространства в трехмерное в процессе аксонометрического преобразования.

## Преобразование списков визуализации в экранные координаты

Преобразование списков визуализации в экранные координаты выполняется аналогично тому, как это делается для **объектов**. Отличается только организация доступа к данным. Сначала рассмотрим версию, в которой предполагается, что ранее произведено нормированное аксонометрическое преобразование с делением на  $z$  (или  $w$ ) и для перехода к экранным координатам нужно выполнить только масштабирование и сдвиг. Приведем код соответствующей функции.

```

void Perspective_To_Screen_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    CAM4OV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
    // Функция преобразует аксонометрические координаты
    // списка визуализации в экранные координаты на основе
    // указанных параметров экрана, содержащихся в объекте
    // cam. Предполагается, что координаты поля обзора
    // нормированы. Эта функция используется вместо функции,
    // основанной на преобразовании объекта, если на
    // предшествующих этапах игрового конвейера все объекты
    // разложены на многоугольники, которые помещаются в
    // глобальный список визуализации. Функция применяется
    // только при условии, что предварительно выполнен

```

```

// переход к нормированным аксонометрическим координатам

// Преобразуем каждый многоугольник в списке
// визуализации из аксонометрических координат в
// экранные. Предполагается, что список визуализации уже
// преобразован в нормированные аксонометрические
// координаты, а результат помещен в массив tvlist[]
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
    // Запрашиваем параметры текущего многоугольника
    POLYF4DV1_PTR curr_poly =
        rend_list->poly_ptrs[poly];

    // Корректен ли данный многоугольник?
    // Многоугольник преобразуется, только если он не
    // отсечен и не отбракован, если он активный и
    // видимый. Однако заметим, что концепция обратной
    // поверхности неприменима в игровом процессоре,
    // работающем в каркасном режиме
    if ((curr_poly == NULL) ||
        !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
        (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
        (curr_poly->state & POLY4DV1_STATE_BACKFACE))
        continue; // Переход к следующему многоугольнику

    float alpha = (0.5*cam->viewport_width - 0.5);
    float beta = (0.5*cam->viewport_height - 0.5);

    // Все в порядке, приступаем к преобразованию
    for (int vertex = 0; vertex < 3; vertex++)
    {
        // Данная вершина задана в нормированных
        // координатах, изменяющихся вдоль каждой оси от
        // -1 до 1. Для проецирования на экран
        // достаточно выполнить масштабирование и
        // поменять направление оси y

        // Преобразуем координаты вершины с помощью
        // параметров камеры
        curr_poly->tvlist[vertex].x = alpha +
            alpha*curr_poly->tvlist[vertex].x;
        curr_poly->tvlist[vertex].y = beta -
            beta *curr_poly->tvlist[vertex].y;
    } // for vertex
} // for poly
} // Perspective_To_Screen_RENDERLIST4DV1

```

Как и следовало ожидать, все свелось к масштабному преобразованию и изменению направления оси y. Приведем пример использования этой функции после того, как была вызвана функция для перехода из координат камеры в аксонометрические координаты, в результате которого координаты стали неоднородными.

```
// Переходим из координат камеры в аксонометрические и делим
// результат на r
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);
```

```
// Отображаем аксонометрические координаты на экран
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);
```

Думаю, вы уже уловили разницу между четырехмерными и трехмерными координатами. Данное преобразование, так же как и для объектов, можно выполнить с помощью матриц. Однако здесь необходимо быть осторожным и учитывать, в какой форме заданы аксонометрические координаты — в четырехмерной или трехмерной. Приведем пример того, как можно все время оставаться в четырехмерном пространстве.

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D(&cam, &mscr);
Transform_RENDERLIST4DV1(&rend_list, &mscr,
    TRANSFORM_TRANS_ONLY);
```

Экранные координаты уже почти получены, осталось только перевести их из однородной формы в неоднородную.

```
Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);
```

Наконец, если координаты заданы уже в трехмерном пространстве, можно забыть об этапе, когда однородные координаты переводятся в неоднородные, и воспользоваться более простым переходом из аксонометрических координат в экранные, с которым мы уже ознакомились. Преобразование с помощью матриц, при котором предполагается, что координаты уже заданы в трехмерном пространстве, выполняется следующим образом.

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);
Transform_RENDERLIST4DV1(&rend_list, &mscr,
    TRANSFORM_TRANS_ONLY);
```

Как видите, преимуществом матриц является то, что их можно применять почти во всех ситуациях. В данном случае единственное различие между обработкой объектов и списков визуализации заключается в последнем вызове функции `Transform_*`, выполняющей это преобразование.

#### СОВЕТ

В реальной жизни я бы оформил функции `Transform_*` как виртуальные (используемые в классах), чтобы можно было ограничиться одним вызовом, в котором осуществлялось бы преобразование объекта. Однако нужно же оставить немного работы и читателю!

## Аксонометрическое преобразование в экранные координаты

Пришло время выполнить небольшую оптимизацию. Я хотел бы продемонстрировать все способы выполнения тех или иных действий, чтобы дать вам ясное представление о том, как работают различные методы. Если вы еще не уснули, читая данную книгу, то к этому моменту должны были бы понять, что матричные операции обладают определенными достоинствами, но — не подходят для выполнения аксонометрического преобразования, и поэтому это преобразование легче выполнить вручную. Кроме того, реализация аксонометрического преобразования и отображения на экран с помощью разных функций во многих случаях вполне обоснована. Однако если изображение нужно просто поместить в буфер, то нет особой необходимости вызывать вторую функцию, потому что

в ней всего-навсего осуществляется масштабирование и сдвиг после аксонометрического преобразования. Эти две операции тривиальны, так что их можно объединить в одну функцию вместе с аксонометрическим преобразованием. В результате мы получим функцию, с помощью которой выполняется как аксонометрическое преобразование, так и преобразование в экранные координаты.

## Преобразование координат камеры в экранные для объектов

Наиболее производительной функцией, выполняющей преобразование координат камеры в экранные, является та, в которой все преобразования, включая деление на  $z$ , выполняются вручную. Для объектов это выполняется следующим образом.

```
void Camera_To_Perspective_Screen_OBJECT4DV1(
    OBJECT4DV1_PTR obj,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
    // Функция преобразует объект, заданный в координатах
    // камеры, в экранные координаты на основе параметров
    // камеры. Функция работает только с вершинами,
    // содержащимися в массиве vlist_trans[]. Кроме прочего,
    // функция меняет направление оси y, поэтому
    // генерируемые ею координаты являются экранными,
    // готовыми для визуализации

    float alpha = (0.5*cam->viewport_width - 0.5);
    float beta = (0.5*cam->viewport_height - 0.5);

    // Преобразуем каждую вершину объекта в
    // аксонометрические координаты. Предполагается, что
    // объект определен в координатах камеры, а результат
    // помещен в массив vlist_trans[]
    for (int vertex = 0; vertex < obj->num_vertices;
        vertex++)
    {
        float r = obj->vlist_trans[vertex].z;

        // Преобразуем вершину с помощью параметров камеры
        obj->vlist_trans[vertex].x = cam->view_dist_h *
            obj->vlist_trans[vertex].x/z;
        obj->vlist_trans[vertex].y = cam->view_dist_v *
            obj->vlist_trans[vertex].y *
            cam->aspect_ratio/z;
        // z = z

        // Заметим, что деление на компоненту w однородных
        // координат НЕ производится, поскольку в этой
        // версии функции матричные операции не используются

        // Теперь значения координат ограничены интервалами:
        // по x от -viewport_width/2 до viewport_width/2,
        // по y от -viewport_height/2 до viewport_height/2,
        // поэтому необходимо выполнить сдвиг. Кроме того,
        // для окончательного перехода к экранным
```

```

// координатам нужно изменить направление оси y
obj->vlist_trans[vertex].x -
    obj->vlist_trans[vertex].x + alpha;
obj->vlist_trans[vertex].y =
    -obj->vlist_trans[vertex].y + beta;

} // for vertex

```

```

} // Camera_To_Perspective_Screen_OBJECT4DV1

```

Заметим, что с помощью этой функции удалось **сэкономить** куда больше математических операций, чем ожидалось. Нам не пришлось выполнять масштабирование при переходе от аксонометрические координат к экранным, потому что проецирование сразу выполняется на поле обзора размерами `viewport_width×viewport_height`, что приводит кдополнительной рационализации. По окончании работы функции координаты вершин, **содержащиеся** в массиве `vlist_trans[]`, будут полностью готовы к выводу на экран. Приведем пример вызова функции.

```

Camera _To_Perspective_Screen_OBJECT4DV1(&obj,&cam);

```

Конечно же, перед тем как вызвать рассмотренную выше функцию, не помешает убедиться, что координаты действительно заданы в системе координат камеры, но в остальном ее применение не приводит ни к каким затруднениям.

### Преобразование координат камеры в экранные для списков визуализации

В преобразовании координат камеры в экранные, выполняемом для списков визуализации, тоже нет ничего необычного — всего лишь немного по-другому организован доступ кструктурамданных. Приведем кодфункции, выполняющейэтузадачу.

```

void Camera_To_Perspective_Screen_RENDERLIST4DV1(
    RENDERLIST4DV1_PTR rend_list,
    CAM4DV1_PTR cam)
{
    // ПРИМЕЧАНИЕ. В этой функции не используются матрицы.
    // Функция преобразует координаты объекта, заданные в
    // системе координат камеры, в экранные координаты на
    // основе параметров камеры. Функция работает с
    // вершинами, содержащимися в массиве tvlist[]. Кроме
    // прочего, в ней изменяется направление оси y, поэтому
    // генерируемые координаты являются экранными, готовыми
    // для визуализации

    // Преобразуем каждый многоугольник в списке
    // визуализации в экранные координаты. Предполагается,
    // что преобразование списка визуализации в координаты
    // камеры уже выполнено, а результат помещен в массив
    // tvlist[]
    for (int poly = 0; poly < rend_list->num_polys; poly++)
    {
        // Запрашиваем параметры текущего многоугольника
        POLYF4DV1_PTR curr_poly=rend_list->poly_ptrs[poly];

        // Корректен ли данный многоугольник?
    }
}

```

```

// Многоугольник преобразуется, только если он не
// отсечен, не отбракован, активный и видимый.
// Однако заметим, что концепция обратных
// поверхностей неприменима в игровом процессоре,
// работающем в каркасном режиме
if ((curr_poly==NULL) ||
    !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE))
    continue; // Переход к следующему многоугольнику

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// Все в порядке, приступаем к преобразованию
for (int vertex = 0; vertex < 3; vertex++)
{
    float z = curr_poly->tvlist[vertex].z;

    // Преобразуем координаты вершин с
    // использованием параметров камеры
    curr_poly->tvlist[vertex].x = cam->view_dist_h *
        curr_poly->tvlist[vertex].x/z;
    curr_poly->tvlist[vertex].y = cam->view_dist_v *
        curr_poly->tvlist[vertex].y *
        cam->aspect_ratio/z;
    // z = z

    // Заметим, что деление на компоненту w
    // однородных координат НЕ выполняется,
    // поскольку в этой версии функции матричные
    // операции не применяются

    // Теперь координаты находятся в интервалах:
    // x: от -viewport_width/2 до viewport_width/2
    // y: от -viewport_height/2 до viewport_height/2
    // Для завершения перехода к экранным
    // координатам надо выполнить сдвиг и изменить
    // направление оси y
    curr_poly->tvlist[vertex].x = curr_poly->
        tvlist[vertex].x + alpha;
    curr_poly->tvlist[vertex].y = -curr_poly->
        tvlist[vertex].y + beta;

} // for vertex

} // for poly

} // Camera_To_Perspective_Screen_RENDERLIST4DV1

```

Эта функция вызывается следующим образом.

```

Camera_To_Perspective_Screen_RENDERLIST4DV1(&rend_list,
    &cam);

```

Вот и все. С основной частью игрового трехмерного конвейера покончено! Возможно, единственной сложностью при выполнении аксонометрического преобразования является принятие решения относительно того, выполнять ли это преобразование вручную или с помощью матриц и следует ли продолжать работу в четырехмерных координатах. С другой стороны, аксонометрическое преобразование и преобразование в экранные координаты можно объединить в один этап, сэкономив при этом много времени и усилий. В большинстве случаев мы так и будем делать. По крайней мере, в игровом конвейере будут использоваться функции, в которых аксонометрическое преобразование и преобразование в экранные координаты выполняется вручную, причем с **последующим** переходом к трехмерным координатам (т.е. **неоднородным**), потому что выполнять преобразование четырехмерных координат в трехмерные на более поздних этапах конвейера нежелательно. Но, как уже было сказано, при непосредственной работе с аппаратным обеспечением не исключена возможность выбора другого **пути**, так как аппаратные решения **чаще** основаны на матричных **операциях** и четырехмерных координатах.

## Визуализация трехмерного игрового мира

Я уже и не надеялся, что мы дойдем до этого **места**... При написании настоящей главы меня смущало то, что я никак не **могу** привести ни одной демонстрационной программы, потому что сначала нужно понять и запрограммировать все этапы игрового конвейера. Однако теперь все готово, и можно создавать такие демонстрационные программы, какие мы только пожелаем — **причем** мы так и поступим. Но давайте сначала еще раз взглянем на трехмерный игровой конвейер, а затем напишем несколько простых графических функций, которые выводят изображения каркасов, заданных объектами **OBJECT4DV1** или **RENDERLIST4DV1**.

### Трехмерный игровой конвейер

Теперь у нас есть весь код, необходимый для загрузки с диска трехмерных объектов, сохранения их в структуре **OBJECT4DV1** и обработки одного или нескольких объектов на всех этапах игрового конвейера. Можно также до определенного момента работать со структурами **OBJECT4DV1**, затем разложить содержащиеся в них объекты на составляющие их многоугольники, после чего поместить их в список, хранящийся в структуре данных **RENDERLIST4DV1**. Как поступать — зависит от вашего желания; оба метода обладают своими сильными и слабыми сторонами. Вообще **говоря**, в большинстве игровых трехмерных процессоров сочетаются оба упомянутых метода. С передвигающимися игровыми элементами они работают как с объектами, а окружающая обстановка, как правило, задается в виде большого набора многоугольников, который разбивается на секторы и на некотором этапе помещается в список визуализации вместе с объектами. Однако в данный момент мы не хотим ничего усложнять, поэтому предполагаем, что все игровые элементы представлены объектами, а окружающая обстановка или интерьер помещений не моделируется.

На рис. 7.19 показаны различные последовательности обработки игровой геометрии. Вообще говоря, какой бы вариант не был выбран, обязательными являются такие этапы:

0. загрузка и размещение объекта;
1. преобразование локальных координат в мировые;
2. удаление лишних объектов и обратных поверхностей;
3. преобразование мировых координат в координаты камеры;

4. преобразование координат камеры в аксонометрические координаты;
5. преобразование аксонометрических координат в экранные;
- в. визуализация геометрии.

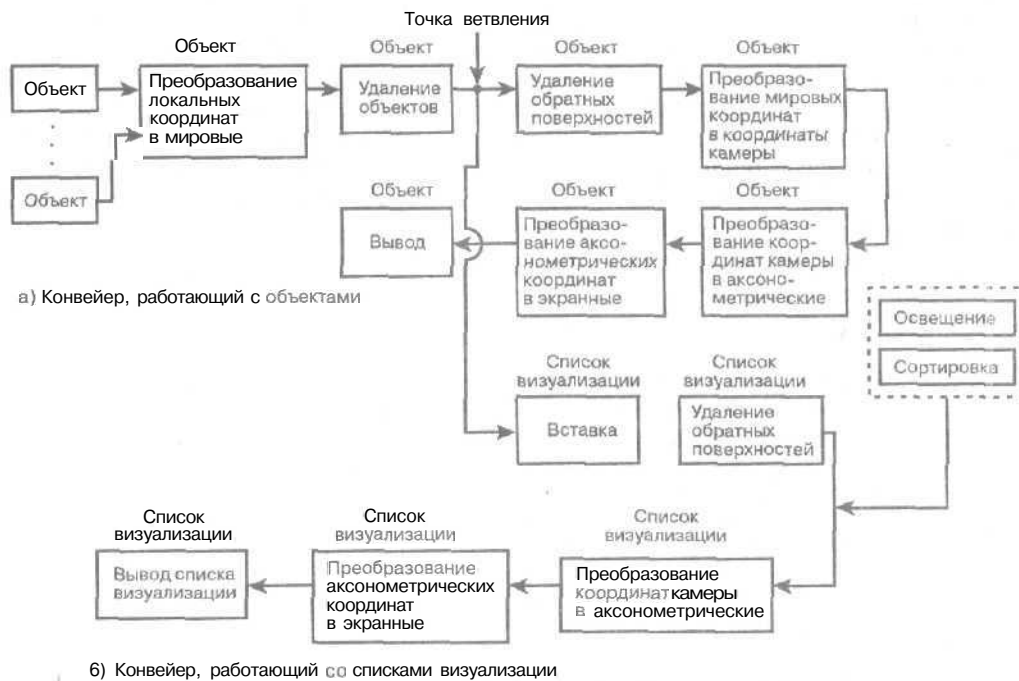


Рис. 7.19. Схема игрового конвейера и различные способы визуализации

Объекты можно разложить на многоугольники на любом этапе конвейера (после второго), потому что удаление объектов удобнее выполнять, пока они целые. Как только объекты преобразованы в набор многоугольников, теряется их структура, и мы уже не можем представлять их как нечто единое. Мы совсем упустили из виду трехмерное отсечение, но после удаления объектов об этой операции можно не думать, поскольку удаление предполагает отсечение по ближней  $z$ -плоскости отсечения. По большому счету, нежелательно иметь дело с геометрическими элементами, содержащими вершины с отрицательными или нулевыми координатами  $z$ , однако это неизбежно. Данный вопрос будет рассмотрен в следующей главе, когда речь пойдет о полном трехмерном отсечении. А пока что предположим, что все спроецированные вершины достаточно безопасны, и если они и *выходят* за рамки двумерного экрана, то этот выход устраняется путем *отсечения* в двумерных функциях вывода.

Независимо от порядка обработки геометрии в нашем трехмерном игровом конвейере, на каждом из его этапов приходится иметь дело с одним из следующих элементов: списком структур `OBJECT4DV1`, содержащих визуализируемые объекты, или единой структурой `RENDERLIST4DV1`, в которой представлены все многоугольники, подлежащие визуализации. Конечно же, можно организовать не один список визуализации, а несколько, но пока что будем предполагать, что все объекты помещены в один список. Теперь нам нужны две функции, одна из которых выводила бы все многоугольники, содержащиеся в структуре `OBJECT4DV1`, а вторая — все многоугольники, содержащиеся в структуре `RENDERLIST4DV1`.

## Вывод содержимого структуры OBJECT4DV1

Прежде чем приступить к разработке функции, напомним, что мы собираемся работать в каркасном режиме без сортировки или удаления скрытых линий. Это означает, что можно будет видеть сквозь объекты. Кроме того, объекты можно выводить в произвольном порядке, потому что их положение по оси z не имеет значения — ведь объекты *пустые*! Поэтому **все**, что нужно, — это создать такую функцию, в которой предполагается, что координаты вершин, содержащиеся в массиве, предназначенном для хранения "преобразованного" списка вершин, заданы корректно и преобразованы к экранным координатам. Функция должна просто получить этот список и вывести все треугольники. Ниже приведен код функции, выполняющей поставленную задачу. Обратите внимание, что в ней **проверяется**, является ли многоугольник видимым, двусторонним, отсеченным и не является ли он обратной поверхностью.

```
void Draw_OBJECT4DV1_Wire16(OBJECT4DV1_PTR obj,
                             UCHAR *video_buffer,
                             int lpitch)
{
    // Эта функция выводит на экран объект в виде каркаса в
    // 16-битовом режиме. В ней не производится удаление
    // скрытых поверхностей и другие подобные операции. Эта
    // функция - простейший способ визуализации объекта без
    // его разложения на отдельные многоугольники. В ней
    // предполагается, что все координаты заданы как
    // экранные (но двумерное отсечение все же выполняется)

    // Выполняем итерации по всему списку многоугольников,
    // входящему в состав объекта, и просто выводим каждый
    // многоугольник
    for (int poly=0; poly < obj->num_polys; poly++)
    {
        // Многоугольник визуализируется только тогда, когда
        // он не отсечен, не отбракован, активный и видимый.
        // Однако заметим, что концепция обратных
        // поверхностей неприменима в игровом процессоре,
        // работающем в каркасном режиме
        if(!(obj->plist[poly].state & POLY4DV1_STATE_ACTIVE) ||
            (obj->plist[poly].state & POLY4DV1_STATE_CLIPPED) ||
            (obj->plist[poly].state & POLY4DV1_STATE_BACKFACE))
            continue; // Переход к следующему многоугольнику

        // Извлекаем индексы вершин из основного списка.
        // Напомним, что многоугольники не являются
        // самодостаточными; они основаны на списке вершин,
        // хранящемся в самом объекте
        int vindex_0 = obj->plist[poly].vert[0];
        int vindex_1 = obj->plist[poly].vert[1];
        int vindex_2 = obj->plist[poly].vert[2];

        // Выводим линии
        Draw_Clip_Line16(obj->vlist_trans[vindex_0].x,
                         obj->vlist_trans[vindex_0].y,
                         obj->vlist_trans[vindex_1].x,
```

```

obj->vlist_trans[ vindex_1 ].y,
obj->plist[poly].color,
video_buffer, lpitch);

Draw_Clip_Line16(obj->vlist_trans[ vindex_1 ].x,
obj->vlist_trans[ vindex_1 ].y,
obj->vlist_trans[ vindex_2 ].x,
obj->vlist_trans[ vindex_2 ].y,
obj->plist[poly].color,
video_buffer, lpitch);

Draw_Clip_Line16(obj->vlist_trans[ vindex_2 ].x,
obj->vlist_trans[ vindex_2 ].y,
obj->vlist_trans[ vindex_0 ].x,
obj->vlist_trans[ vindex_0 ].y,
obj->plist[poly].color,
video_buffer, lpitch);

} // for poly

} // Draw_OBJECT4DV1_Wire

```

Замечательная функция, не правда ли? Располагая виртуальной компьютерной системой и графическим буфером, можно заниматься созданием трехмерных функций, ничего не зная об аппаратном обеспечении. Все, что нам нужно, — адрес видеобуфера и шаг памяти. В этой функции применяется несколько двумерных функций, предназначенных для вывода линий. В ней просто последовательно обрабатываются элементы списка многоугольников и выводятся все треугольники, из которых состоит трехмерный объект. Ниже приведен пример вызова функции в предположении, что все параметры графической системы уже настроены, все предварительные этапы трехмерного игрового конвейера пройдены, а объект obj готов к дальнейшему применению,

```
Draw_OBJECT4DV1_Wire16(&obj, video_buffer, lpitch);
```

## Вывод содержимого структуры RENDERLIST4DV1

Функция вывода списка визуализации идентична работающей с объектами, однако в ней по-другому организован доступ к структуре данных. Ниже представлен ее код.

```

void Draw_RENDERLIST4DV1_Wire16(
    RENDERLIST4DV1_PTR rend_list,
    UCHAR *video_buffer, int lpitch)
{
    // Эта функция обрабатывает список визуализации, другими
    // словами, она выводит все поверхности, хранящиеся в
    // списке, в виде каркаса в 16-битовом режиме. Заметим,
    // что пока нет необходимости сортировать многоугольники
    // каркаса (хотя позже эта возможность понадобится). На
    // функцию возлагается задача правильно определить
    // битовую глубину и вызвать соответствующую функцию
    // растеризации

    // Все, что нужно в данный момент, - это список
    // многоугольников; пришло время вывести его на экран

```

```

for (int poly=0; poly < rend_list->num_polys; poly++)
{
    // Многоугольник визуализируется, только если он не
    // отсекается, не отбраковывается, если он активный
    // и видимый. Однако заметим, что концепция обратных
    // поверхностей неприменима в игровых процессорах,
    // работающих в каркасном режиме
    if (!(rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_ACTIVE) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_CLIPPED) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_BACKFACE) )
        continue; // Переход к следующему многоугольнику

    // Выводим стороны треугольника. Заметим, что
    // параметры отсечения уже настроены в процессе
    // инициализации двумерной подсистемы, так что
    // функция отсечения произведет усечение всех
    // многоугольников, выходящих за рамки двумерной
    // области экрана
    Draw_Clip_Line16(
        rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);

    Draw_Clip_Line16(
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);

    Draw_Clip_Line16(
        rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);

} // for poly

} // Draw_RENDERLIST4DV1_Wire

Способ вызова этой функции не отличается от вызова предыдущей, только вместо объек-
та в нее нужно передать список визуализации.
Draw_RENDERLIST4DV1_Wire16(&rend_list, video_buffer, lpitch);

```

Вот и все. Теперь мы готовы к знакомству с демонстрационными программами. Сейчас, когда работа над игровым конвейером и алгоритмами визуализации завершена, можно приступить к разработке нескольких демонстрационных программ, чтобы получить практические навыки работы с различными методами, описанными в предыдущих разделах. В следующем разделе рассматриваются примеры, дающие представление о том, как все описанное работает в реальном мире.

## Трехмерные демонстрационные программы

Рассмотрим несколько завершенных примеров. Пока мы не будем создавать ничего слишком сложного. Мы собираемся ограничиться простейшими примерами построения каркасов, основанными на применении разработанных ранее функций. Но, как уже было сказано, после прочтения этой главы вы уже сможете написать игру, подобную *Battle Zone*. Каждая демонстрационная программа снабжена заголовком и рядом комментариев, описывающих происходящее. Если хотите, данный раздел можно опустить и сразу перейти к изучению тех вопросов, которые вас интересуют. Кроме того, я не стану полностью приводить исходный код всех демонстрационных программ, поскольку мы уже ознакомились со всеми используемыми в них функциями. В книгу включены только наиболее важные фрагменты основного цикла игр. Чтобы скомпилировать любую из этих демонстрационных программ, понадобятся все файлы, список которых приведен ниже, а также основной исходный файл с расширением .CPP.

```
T3DLIB1.CPP|H;  
T3DLIB2.CPP|H;  
T3DLIB3.CPP|H;  
T3DLIB4.CPP|H;  
T3DLIB5.CPP|H;  
DDRAW.LIB;  
DSOUND.LIB;  
DINPUT.LIB;  
DINPUT8.LIB;
```

Кроме того, конечно же, будут нужны все файлы с расширением .PLG, которые содержатся в папке данной главы.

## Вывод отдельного треугольника

**Загружаемые объекты:** отсутствуют; единственный имеющийся многоугольник создается вручную.

**Тип камеры:** эйлерова камера с фиксированным положением, углом обзора 90° и нормированным полем обзора.

**Тип проекции:** аксонометрическая с последующим отображением на экран.

**Математические преобразования, используемые в трехмерном игровом конвейере:** вызываются функции, в которых все преобразования выполняются вручную; матрицы нигде не используются.

**Удаление обратных поверхностей:** отсутствует.

**Удаление объектов:** отсутствует.

**Тип визуализируемой геометрии:** список визуализации, в котором только один многоугольник.

Демонстрационная программа `DEMO17_1.CPP|EXE` настолько проста, насколько это возможно. В ней создается только один многоугольник, вращающийся вокруг оси у, и все! Приведем список важных глобальных переменных, используемых в этой программе,

```
// Инициализируем положение и направление камеры
POINT4D cam_pos - {0,0,-100,1};
VECTOR4D cam_dir - {0,0,0,1};
```

```
// Вот весь код инициализации...
VECTOR4D vscale={0.5,0.5,0.5,1}, vpos = {0,0,0,1},
          vrot = {0,0,0,1};
```

```
CAM4DV1 cam; // Одна камера
RENDERLIST4DV1 rend_list; // Один список визуализации
POLYF4DV1 poly1; // Единственный многоугольник
POINT4D poly1_pos =
    {0,0,100,1}; // Положение многоугольника в
                // мировых координатах
```

Приведен также код функции `Game_Init()`.

```
// Инициализируем математический процессор
Build_Sin_Cos_Tables();
```

```
// Инициализируем единственный имеющийся многоугольник
poly1.state = POLY4DV1_STATE_ACTIVE;
poly1.attr = 0;
poly1.color = RGB16Bit(0,255,0);
```

```
poly1.vlist[0].x = 0;
poly1.vlist[0].y = 50;
poly1.vlist[0].z = 0;
poly1.vlist[0].w = 1;
```

```
poly1.vlist[1].x = 50;
poly1.vlist[1].y = -50;
poly1.vlist[1].z = 0;
poly1.vlist[1].w = 1;
```

```
poly1.vlist[2].x = -50;
poly1.vlist[2].y = -50;
poly1.vlist[2].z = 0;
poly1.vlist[2].w = 1;
```

```
poly1.next = poly1.prev = NULL;
```

```
// Инициализируем параметры камеры с углом обзора 90° и
// нормированными координатами поля обзора
Init_CAM4DV1(&cam, // Объект, представляющий камеру
             &cam_pos, // Начальное положение камеры
             &cam_dir, // Начальная ориентация камеры
             50.0, // Ближняя и дальняя
             500.0, // плоскостиотсечения
             90.0, // Угол обзора в градусах
             1.0, // Расстояние до точки наблюдения
             0, // Уменьшенные на единицу размеры
             0, // поля обзора
```

```
WINDOW_WIDTH, // Размер экрана
WINDOW_HEIGHT);
```

Наконец, приведем основную функцию `Game_Main()`, в которой реализован весь трехмерный игровой конвейер и визуализация треугольника.

```
int Game_Main(void *parms)
{
    // Эта функция - "рабочая лошадка" игры. Она постоянно
    // вызывается в реальном времени и аналогична функции
    // main() в языке C. Из нее вызываются все остальные
    // функции игры

    static MATRIX4X4 mrot; // Матрица поворота общего
                           // назначения
    static float ang_y = 0; // Угол поворота

    int index;             // Переменная цикла

    // Запускаем таймер
    Start_Clock();

    // Очищаем поверхность вывода
    DDDraw_Fill_Surface(lpddsback, 0);

    // Здесь считываем информацию, поступающую с клавиатуры
    // и других устройств
    DInput_Read_Keyboard();

    // Здесь выполняется логика игры...

    // Инициализируем список визуализации
    Reset_RENDERLIST4DV1(&rend_list);

    // Помещаем многоугольник в список визуализации
    Insert_POLYF4DV1_RENDERLIST4DV1(&rend_list, &poly1);

    // Генерируем матрицу поворота вокруг оси y
    Build_XYZ_Rotation_MATRIX4X4(0, ang_y, 0, &mrot);

    // Медленно вращаем многоугольник
    if (++ang_y >= 360.0) ang_y = 0;

    // Организуем вращение локальных координат единственного
    // многоугольника, имеющегося в списке визуализации
    Transform_RENDERLIST4DV1(&rend_list, &mrot,
                             TRANSFORM_LOCAL_ONLY);

    // Переходим из локальных координат модели в мировые
    Model_To_World_RENDERLIST4DV1(&rend_list, &poly1_pos);

    // Генерируем матрицу камеры
    Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

```

// Переходим из мировых координат в координаты камеры
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// Переходим из координат камеры в аксонометрические
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// Переходим в экранные координаты
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// Блокируем вторичный буфер
DDraw_Lock_Back_Surface();

// Визуализируем список многоугольников
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer,
                             back_lpitch);

// Снимаем блокировку вторичного буфера
DDraw_Unlock_Back_Surface();

// Переключение поверхностей
DDraw_Flip();

// Синхронизация с частотой кадров 30 fps
Wait_Clock(30);

// Проверяем, не намерен ли пользователь выйти из игры
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
{
    PostMessage(main_window_handle, WM_DESTROY, 0, 0);
} // if

// Код успешного выполнения
return(1);

} // Game_Main

```

На рис 7.20 приведена копия экрана, полученная в процессе работы программы DEMOIII\_1.EXE. Основное предназначение этой демонстрационной программы — показать, что, вызвав всего около десятка функций, можно создать полноценное изображение трехмерного каркаса.

## Вывод трехмерного каркаса куба

**Загружаемые объекты:** обычный куб (его модель содержится в файле CUBE1.PLG).

**Тип камеры:** эйлерова камера с фиксированным положением, углом обзора 90°, нормированным полем обзора 2x2 и экраном 400x400.

**Тип проекции:** аксонометрическая с последующим отображением на экран,

**Математические преобразования, используемые в трехмерном игровом конвейере:** вызываются функции, в которых все преобразования выполняются вручную; матрицы нигде не используются.

**Удаление обратных поверхностей:** отсутствует.

**Удаление объектов:** отсутствует.

**Тип визуализируемой геометрии:** визуализация объекта; список визуализации отсутствует.

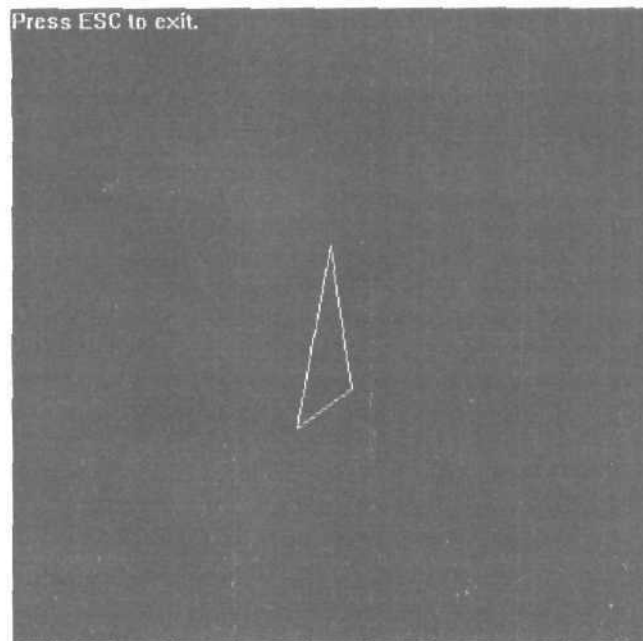


Рис. 7.20. Копия экрана, полученная в ходе работы программы DEMOII7\_1.EXE

Теперь, когда мы ознакомились с примером, в котором принимает участие единственный вращающийся многоугольник, давайте сделаем кое-что поинтереснее. Демонстрационная программа DEMOII7\_2.CPP|EXE аналогична предыдущей, но на этот раз в ней применяется возможность загрузки **моделей**, а именно — загружается каркас куба из файла CUBE1.PLG. Содержимое файла приведено ниже.

# Начало файла plg/plx.

# Обычный куб.  
tri 8 12

# Список вершин.  
5 5 5  
-5 5 5  
-5 5 -5  
5 5 -5  
5 -5 5  
-5 -5 5  
-5 -5 -5  
5 -5 -5

# **Список** многоугольников.  
0xd0f0 3 2 1 0  
0xd0f0 3 3 2 0  
0xd0f0 3 4 7 0  
**0xd0f0** 3 7 3 0

```

0xd0f0 3 6 7 4
0xd0f0 3 5 6 4
0xd0f0 3 2 6 1
0xd0f0 3 6 5 1
0xd0f0 3 7 6 3
0xd0f0 3 6 2 3
0xd0f0 3 5 4 0
0xd0f0 3 1 5 0

```

Перед вами модель куба 10x10x10. После загрузки каркаса в структуру OBJECT4DV1 он медленно **вращается**. Вращение куба и его **последующие** преобразования трехмерным игровым конвейером, предшествующие выводу на экран, реализованы в основном цикле программы. Следует отметить, что куб не преобразуется в набор **многоугольников**, которые затем можно было бы поместить в список визуализации. Объект остается цельным кубом. Это позволяет продемонстрировать, что целостность объекта можно сохранять на протяжении всех этапов игрового конвейера.

Приведем объявления основных глобальных переменных, **использующихся** в демонстрационной программе.

```

//Инициализируем положение и направление камеры
POINT4D cam_pos = {0,0,0,1};
VECTOR4D cam_dir = {0,0,0,1};

```

```

// Здесь идет код инициализации...
VECTOR4D vscale={5.0,5.0,5.0,1}, vpos = {0,0,0,1},
          vrot = {0,0,0,1};

```

```

CAM4DV1 cam; // Единственная камера
OBJECT4DV1 obj; //Хранилище каркаса куба

```

Обратите внимание на величину vscale. Этот вектор осуществляет масштабирование каркаса в **процессе** его загрузки, поэтому куб 10x10x10 в ходе загрузки превращается в куб 50x50x50. Конечно же, можно было изменить саму модель, однако здесь **главный** смысл в гибкости, которой обладает **функция** загрузки. Возможность масштабирования объекта во время загрузки вносит дополнительные возможности работы с каркасом.

А теперь ознакомимся с той частью функции Game\_Init(), в которой производится инициализация. Она аналогична **соответствующей** части предыдущей **функции**, однако **здесь** загружается объект:

```

Init_CAM4DV1(&cam, //Объект,представляющийкамеру
             &cam_pos, // Начальное положение камеры
             &cam_dir, // Начальная ориентация камеры
             50.0, // Ближняя и дальняя плоскости отсечения
             500.0,
             90.0, // Поле обзора в градусах
             WINDOW_WIDTH, // Размер экрана
             WINDOW_HEIGHT);

```

```

// Загружаем куб
Load_OBJECT4DV1_PLG(&obj, "cube1.plg",
                   &vscale, &vpos, &vrot);

```

```

// Задаем положение куба в мировых координатах
obj.world_pos.x = 0;

```

```
obj.world_pos.y = 0;
obj.world_pos.z = 100;
```

Обратите внимание, что объект располагается на расстоянии 100 единиц от начала координат (в мировой системе координат). Это сделано для того, чтобы он казался достаточно большим и чтобы была возможность показать перспективу. Наконец, приведем код функции `Game_Main()`, чтобы иметь возможность ознакомиться с вызовами функций в игровом конвейере.

```
int Game_Main(void *parms)
{
    // Эта функция - "рабочая лошадка" игры. Она постоянно
    // вызывается в реальном времени и аналогична функции
    // main() в языке С. Здесь вызываются все остальные
    // функции

    static MATRIX4X4 mrot; // Матрица поворота общего
                           // назначения
    int index;             // Переменная цикла

    // Запускаем таймер
    Start_Clock();

    // Очищаем поверхность вывода
    DDDraw_Fill_Surface(lpddsback, 0);

    // Считываем информацию, поступающую с клавиатуры и от
    // других устройств
    DInput_Read_Keyboard();

    // Здесь выполняется логика игры...

    // Сбрасываем состояние объекта (это относится только к
    // обратным поверхностям и удалению объекта)
    Reset_OBJECT4DV1(&obj);

    // Генерируем матрицу поворота вокруг оси y
    Build_XYZ_Rotation_MATRIX4X4(0,5, 0, &mrot);

    // Выполняем поворот объекта
    Transform_OBJECT4DV1(&obj, &mrot,
        TRANSFORM_LOCAL_ONLY,1);

    // Переходим из локальных координат модели к мировым
    Model_To_World_OBJECT4DV1(&obj);

    // Генерируем матрицу камеры
    Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

    // Переходим из мировых координат в координаты камеры
    World_To_Camera_OBJECT4DV1(&obj, &cam);

    // Переходим из координат камеры в аксонометрические
    // координаты
```

```

Camera_To_Perspective_OBJECT4DV1(&obj, &cam);

// Переходим в экранные координаты
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);

// Блокируем вторичный буфер
DDraw_Lock_Back_Surface();

// Визуализируем объект
Draw_OBJECT4DV1_Wire16(&obj, back_buffer, back_pitch);

// Снимаем блокировку вторичного буфера
DDraw_Unlock_Back_Surface();

// Переключение поверхностей
DDraw_Flip();

// Синхронизация с частотой кадров 30 fps
Wait_Clock(30);

// Проверяем, не намерен ли пользователь выйти из игры
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
{
    PostMessage(main_window_handle, WM_DESTROY, 0, 0);
} // if

// Код успешного выполнения,
return(1);

} // Game_Main

```

Как и в предыдущей функции, всего около дюжины обращений к функциям — и трехмерная демонстрационная программа готова! На рис. 7.21 приведена копия экрана, полученная в ходе работы программы `ДЕМОП7_2.EXE`. Обратите внимание на то, как хорошо проявляется перспектива. Это происходит благодаря удачному выбору параметров поля обзора и расстояния до точки наблюдения.

## Вывод трехмерного каркаса куба с удалением обратных поверхностей

**Загружаемые объекты:** обычный куб (его модель содержится в файле `CUBE2.PLG`).

**Тип камеры:** эйлерова камера с фиксированным положением, углом обзора  $90^\circ$ , нормированным полем обзора от  $-1/\text{ar}$  до  $1/\text{ar}$  и экраном  $640 \times 480$ .

**Тип проекции:** аксонометрическая с последующим отображением на экран.

**Математические преобразования, используемые в трехмерном игровом конвейере:** вызываются функции, в которых все преобразования выполняются вручную; матрицы нигде не используются.

**Удаление обратных поверхностей:** производится.

**Удаление объектов:** отсутствует.

**Тип визуализируемой геометрии:** визуализация объекта; список визуализации отсутствует.

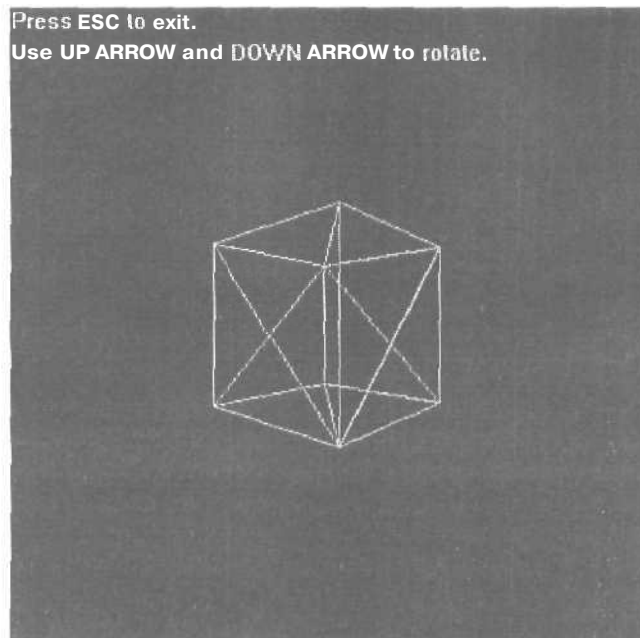


Рис. 7.21. Копия экрана, полученная в ходе работы программы DEMO17\_2.EXE

Демонстрационная программа DEMO17\_3.CPP|EXE идентична программе DEMO17\_2.CPP|EXE. Немногие отличия заключаются в том, что в ней производится удаление обратных поверхностей, а также задаются большие размеры экрана. Для удаления обратных поверхностей следует не забывать о некоторых вещах. Первое — это сам каркас модели. Напомним, что при загрузке каркаса многоугольник может быть идентифицирован как односторонний или двусторонний. Для односторонних многоугольников имеет смысл производить удаление обратных поверхностей. Если же многоугольник двусторонний, то такая операция теряет смысл, потому что он видим с обеих сторон. Очевидно, что сейчас, когда игровой процессор работает в каркасном режиме, удаление обратных поверхностей не играет большой роли, но при желании его все же можно включить. Второй момент, который нужно учитывать, заключается в том, что удаление обратных поверхностей следует выполнять после перехода к мировым координатам, но перед переходом к координатам камеры.

Сначала проверим, есть ли возможность производить удаление обратных поверхностей в каркасе, содержащемся в файле CUBE1.PLG. Приведем одну из строк с дескриптором многоугольника.

```
# Список многоугольников.
0xd0f0 3 2 1 0 # polygon 0
```

Первое число (0xd0f0) — это дескриптор многоугольника. Как было сказано в разделе, где описывался процесс создания загрузчика файлов в формате .PLG/.PLX, бит D дескриптора поверхности указывает на то, является ли многоугольник двусторонним.

```
CSSD | RRRR | GGGG | BBBB
```

В нашем случае этот бит установлен, и его необходимо снять. Для этого нужно создать еще один файл в формате .PLG; дадим ему имя CUBE2.PLG. Этот файл содержится на прилагаемом компакт-диске, и в нем сброшен флаг двусторонней поверхности, поэтому де-

скриптор имеет значение **1100b**, или **0xС** в **шестнадцатеричной** системе счисления. Дескриптор многоугольника теперь выглядит следующим образом.

```
# Список многоугольников.  
0xc0f0 3 2 10 # Многоугольник 0.
```

Чтобы удалить обратные поверхности, нужно вызвать соответствующую **функцию**. Это делается так:

```
// Удаление обратных поверхностей  
Remove_Backfaces_OBJECT4DV1(&obj, &cam);
```

Этот вызов помещается после создания матрицы камеры для заданных углов обзора, но до перехода из мировых координат в координаты камеры. Таким образом удастся избежать преобразования невидимых многоугольников при переходе в систему координат камеры.

На рис. 7.22 приведена копия экрана, полученная в ходе работы программы **DEMOII7\_3.EXE**. Понажимайте клавиши со стрелками "вверх" и "вниз" и наблюдайте за тем, как куб вращается вокруг оси x.

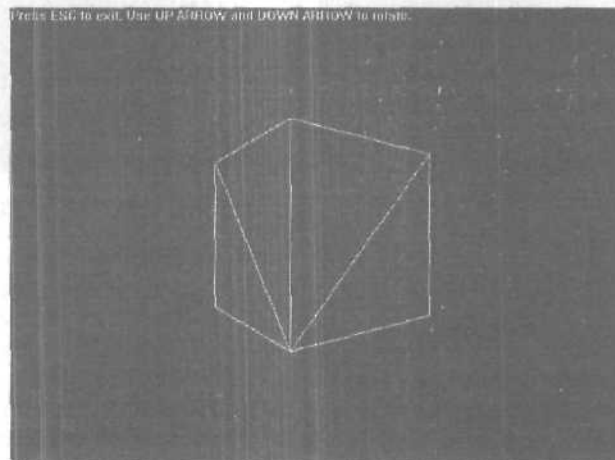


Рис. 7.22. Копия экрана, полученная в ходе работы программы **DEMOII7\_3.EXE**

## Трехмерные танки

**Загружаемые объекты:** простой каркас танка (его модель содержится в файле **TANK1.PLG**).

**Тип камеры:** эйлерова камера с возможностью вращения, углом обзора **90°**, нормированным полем обзора от **-1/аg** до **1/аg** и экраном **400x400**.

**Тип проекции:** аксонометрическая с последующим отображением на экран.

**Математические преобразования, используемые в трехмерном игровом конвейере:** вызываются функции, в которых все преобразования выполняются вручную; матрицы нигде не используются.

**Удаление обратных поверхностей:** отсутствует.

**Удаление объектов:** производится.

**Тип визуализируемой геометрии:** список визуализации, содержащий несколько объектов.

На рис. 7.23 приведена копия экрана, полученная в ходе работы программы **DEMOII7\_4.CPP**. Наконец-то мы видим на экране не **какие-то** геометрические фигуры, а нечто, напоминающее танки. Между прочим, это довольно сложная демонстрационная

программа. Параметры камеры заданы как обычно, модели танков загружаются, а действие игры разворачивается в основном цикле.

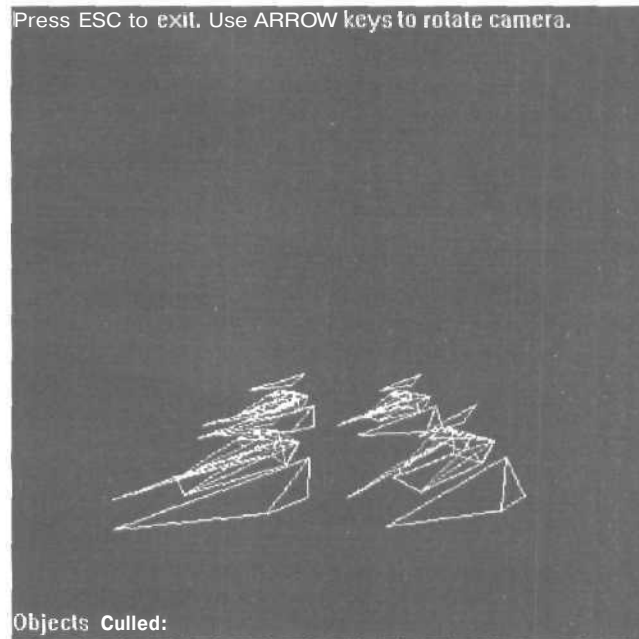


Рис. 7.23. Копия экрана, полученная в ходе работы программы DEMOII7\_4.EXE

Для создания нескольких копий танка в демонстрационной программе применяется один интересный прием. Вместо того чтобы загружать несколько танков, просто изменяется положение "главного" танка, а затем он несколько раз помещается в список визуализации, в результате чего создается видимость наличия нескольких танков. Ниже приведен отрывок программы, содержащий важные части ее основного цикла; код, в котором осуществляется копирование танка и помещение его "клонов" в список визуализации, выделен полужирным шрифтом.

```
intGame_Main(void *parms)
{
    // Эта функция - "рабочая лошадка" игры. Она постоянно
    // вызывается в реальном времени и аналогична функции
    // main() в языке С. Здесь вызываются все остальные
    // функции

    static MATRIX4X4 mrot; // Матрица поворота общего
        // назначения
    static float x_ang = 0, y_ang = 5, z_ang = 0;
    char work_string[256];

    int index; // Переменная цикла

    // Запускаем таймер
    Start_Clock();
```

```

// Очищаем поверхность вывода
DDraw_Fill_Surface(lpddsback, 0);

// Здесь считываем информацию, поступающую с клавиатуры
// и от других устройств
DInput_Read_Keyboard();

// Здесь выполняется логика игры...

// Сбрасываем параметры списка визуализации
Reset_RENDERLIST4DV1(&rend_list);

// Сбрасываем углы
x_ang = 0;
y_ang = 1;
z_ang = 0;

// Проверяем, не пытается ли пользователь вращать камеру
if (KEY_DOWN(VK_DOWN))
    cam.dir.x+=1;
else
    if (KEY_DOWN(VK_UP))
        cam.dir.x-=1;
    if (KEY_DOWN(VK_RIGHT))
        cam.dir.y+=1;
    else
        if (KEY_DOWN(VK_LEFT))
            cam.dir.y+=1;

// Генерируем матрицу поворота камеры вокруг оси y
Build_XYZ_Rotation_MATRIX4X4(x_ang,y_ang,z_ang,&mrot);
// Реализуем поворот локальных координат объекта
Transform_OBJECT4DV1(&obj,&mrot,TRANSFORM_LOCAL_ONLY,1);

// Отбраковываем текущий объект
strcpy(buffer,"Objects Culled: ");

for (intx=-NUM_OBJECTS/2; x < NUM_OBJECTS/2; x++)
    for (int z=-NUM_OBJECTS/2; z < NUM_OBJECTS/2; z++)
    {
        // Сбрасываем параметры объекта (это относится
        // только к обратным поверхностям и удалению
        // объекта)
        Reset_OBJECT4DV1 (&obj);

        // Задаем положение объекта
        obj.world_pos.x = x*OBJECT_SPACING+
            OBJECT_SPACING/2;
        obj.world_pos.y = 0;
        obj.world_pos.z = 500+z*OBJECT_SPACING+
            OBJECT_SPACING/2;
    }

```

```

// Проверяем, нужно ли отбраковать объект
if (!Cull_OBJECT4DV1 (&obj, &cam,
    CULL_OBJECT_XYZ_PLANES))
{
    // Если мы дошли до этого места, то
    // расположенный в данной точке объект видим
    // в мировой системе координат, и его можно
    // помещать в список визуализации

    // Переходим из локальных координат модели в
    // мировые координаты
    Model_To_World_OBJECT4DV1 (&obj);

    // Помещаем объект в список визуализации
    Insert_OBJECT4DV1_RENDERLIST4DV1 (
        ftrencllistf&obj);
} //if
else
(
    sprintf(work_string, "[%d, %d] ", x,z);
    strcat(buffer, work_string);
)

} // for

Draw_Text_GDI(buffer, 0, WINDOW_HEIGHT-20, RGB(0,255,0),
    lpddsback);

// Генерируем матрицу камеры
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// Удаляем обратные поверхности
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);

// Переходим из мировых координат в координаты камеры
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// Переходим из координат камеры в аксонометрические
// координаты
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// Переходим в экранные координаты
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// Вывод на экран
Draw_Text_GDI("Press ESC to exit. Use ARROW "
    "keys to rotate camera.",
    0, 0, RGB(0,255,0), lpddsback);

// Блокируем вторичный буфер
DDraw_Lock_Back_Surface();

```

```

// Визуализируем объект
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer,
                             back_lpitch);

// Снимаем блокировку вторичного буфера
DDraw_Unlock_Back_Surface();

// Переключение поверхностей
DDraw_Flip();

// Синхронизация частоты кадров 30 fps
Wait_Clock(30);

// Проверяем, не пытается ли пользователь выйти из игры
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
{
    PostMessage(main_window_handle, WM_DESTROY, 0, 0);
} // if

// Код успешного выполнения
return(1);

} // Game_Main

```

Теперь **понятно**, почему так важно иметь в своем распоряжении список визуализации и почему это лучший способ визуализации, чем визуализация отдельных объектов. С объектами удобно работать, когда выполняются различные преобразования, анимация, производятся действия, связанные и искусственным интеллектом и логикой игры и т.д. Но как только начинается стадия визуализации, лучше разложить объекты на отдельные многоугольники (после отбраковки объектов) и поместить их в основной список.

## Трехмерные танки и перемещаемая камера

**Загружаемые объекты:** простой каркас танка (его модель содержится в файле TANK1.PLG).

**Тип камеры:** модель UVN с возможностью вращения, углом обзора 90°, нормированным полем обзора от -1/аgдо 1/аgи экраном 800х600 (в полноэкранный режим).

**Тип проекции:** аксонометрическая с последующим отображением на экран.

**Математические преобразования, используемые в трехмерном игровом конвейере:** вызываются функции, в которых все преобразования выполняются вручную; матрицы нигде не используются.

**Удаление обратных поверхностей:** отсутствует.

**Удаление объектов:** производится.

**Тип визуализируемой геометрии:** список визуализации, содержащий несколько объектов.

На рис. 7.24 приведен снимок экрана самой сложной из приводившихся ранее программ. Я бросил в атаку целую армию клонов! В этой демонстрационной программе используется UVN-модель камеры, поэтому единственное, что действительно отличает ее от других программ, — это вызов функции Build\_CAM4DV1\_Matrix\_UVN() вместо обращения к аналогичной функции для эйлеровой камеры. Кроме того, в рассматриваемой программе используется параметр UNV\_MODE\_SIMPLE, с помощью которого в функцию, генерирующую матрицу камеры, передается информация о том, что положение целевой точки задается вручную.





миться, и добавлена парочка программистских трюков. В демонстрационной программе принимают участие объекты трех типов: башни, танки и наземные маркеры. Поскольку визуализация производится в каркасном режиме, трудно создать ощущение глубины, и поэтому мне пришлось добавить большое количество маленьких пирамид, расположенных по всей игровой арене для создания ощущения проекции. Кроме того, в игре имитируется небо и земля. Это сделано с помощью двух заполненных прямоугольников, представляющих собой зеркальное отражение друг друга (рис. 7.26).

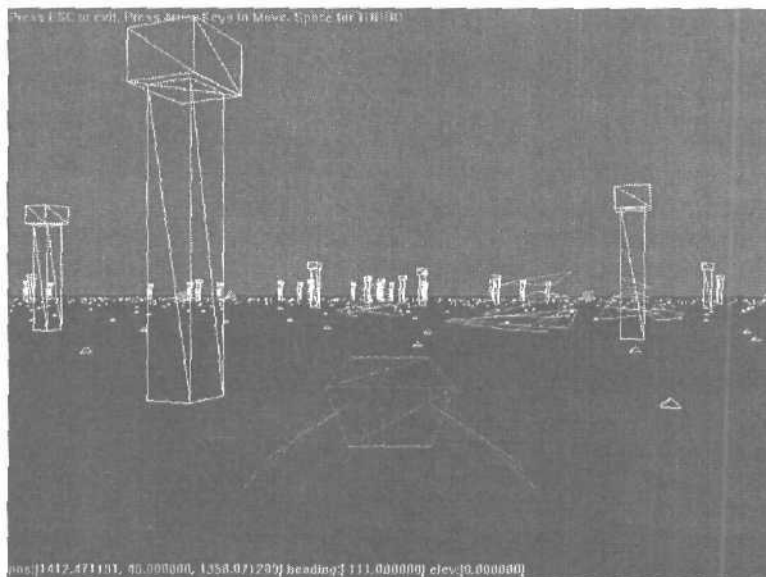


Рис. 7.26. Копия экрана, полученная входе работы программы DEMO17\_6.EXE

Важной особенностью в рассматриваемом примере является то, что в нем еще раз наглядно демонстрируется, как легко создаются копии одного и того же объекта путем его сдвига с последующим размещением преобразованной версии объекта в списке визуализации. Исходный объект при этом не разрушается. Пользуясь данным приемом, конечно же, нужно следить за расположением и ориентацией объектов-копий, но это легко делается с помощью нескольких массивов, элементами которых являются векторы. Рассмотрим теперь различные элементы демонстрационной программы и принцип их работы. Для иллюстрации изложения будут приводиться небольшие отрывки кода из демонстрационной программы, так как весь ее листинг занял бы слишком много места.

### Загрузка и инициализация объектов

Как уже было сказано, в игре участвуют объекты трех типов; танки, башни и маркеры. Все объекты определенного вида созданы с помощью одного исходного, который я называю *главным объектом* (master object). Для хранения параметров, отличающих одну копию от другой (в данном случае это просто координаты, задающие положение объекта, потому что все они неподвижны), были созданы вторичные структуры данных. Приведем фрагмент кода, реализующего загрузку объектов и присвоение им параметров. Обратите внимание на масштабирование, которому подвергается каждый каркас; иногда приходится прибегать к такого рода приемам.

```

// Загружаем главный объект, представляющий танк
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_tank, "tank2.plg",
    &vscale, &vpos, &vrot);

// Загружаем объект, с помощью которого игрок следит за
// происходящим
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_player, "tank3.plg", &vscale,
    &vpos, &vrot);

// Загружаем главный объект, представляющий башню
VECTOR4D_INITXYZ(&vscale,1.0, 2.0, 1.0);
Load_OBJECT4DV1_PLG(&obj_tower, "tower1.plg",
    &vscale, &vpos, &vrot);

// Загружаем главный объект, представляющий маркер
VECTOR4D_INITXYZ(&vscale,3.0,3.0,3.0);
Load_OBJECT4DV1_PLG(&obj_marker, "marker1.plg",
    &vscale, &vpos, &vrot);

// Размещаем танки
for (index = 0; index < NUM_TANKS; index++)
{
    i
    // Положения танков задаются случайным образом
    tanks[index].x = RAND_RANGE(-UNIVERSE_RADIUS,
        UNIVERSE_RADIUS);
    tanks[index].y = 0;
    tanks[index].z = RAND_RANGE(-UNIVERSE_RADIUS,
        UNIVERSE_RADIUS);
    tanks[index].w = RAND_RANGE(0,360);
} // for

// Размещаем башни
for (index = 0; index < NUM_TOWERS; index++)
{
    i
    // Положения башен задаются случайными числами
    towers[index].x = RAND_RANGE(-UNIVERSE_RADIUS,
        UNIVERSE_RADIUS);
    towers[index].y = 0;
    towers[index].z = RAND_RANGE(-UNIVERSE_RADIUS,
        UNIVERSE_RADIUS);
} // for

```

## Камера

Камера моделируется очень просто. Для этого используется эйлерова модель, которая способна перемещаться по игровому полю и вращаться вокруг оси  $y$ . Можно было бы также добавить возможность изменять угол наклона камеры относительно оси  $x$  (угол возвышения), но горизонт в этой программе является искусственным, и нам пришлось бы определять, как при изменении угла **возвышения** изменяются прямоугольники, создающие искусственный горизонт, а это слишком утомительно! Итак, приведем фрагмент кода, в котором реализовано перемещение камеры.

```

// Добавляем возможность перемещения камеры

// Турборежим
if (keyboard_state[DIK_SPACE])
    tank_speed = 5*TANK_SPEED;
else
    tank_speed = TANK_SPEED;

// Движение вперед и назад
if (keyboard_state[DIK_UP])
{
    // Движение вперед
    cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);
    cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
} // if

if (keyboard_state[DIK_DOWN])
{
    // Движение назад
    cam.pos.x -= tank_speed*Fast_Sin(cam.dir.y);
    cam.pos.z -= tank_speed*Fast_Cos(cam.dir.y);
} // if

// Вращение
if (keyboard_state[DIK_RIGHT])
{
    cam.dir.y+=3;

    // Поворачиваем объект игрока
    if ((turning+=2) > 15)
        turning=15;
} // if

if (keyboard_state[DIK_LEFT])
{
    cam.dir.y-=3;

    // Поворачиваем объект игрока
    if ((turning-=2) < -15)
        turning=-15;
} // if
else // Возврат к нулевому углу направления игрока
{
    if (turning > 0)
        turning-=1;
    else
        if (turning < 0)
            turning+=1;
} // else

// Генерируем матрицу камеры
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

```

По сути, это весь код, моделирующий камеру. Он довольно прост. В зависимости от того, какие клавиши нажаты, камера перемещается вдоль вектора, задающего ее направление. При этом не следует забывать, что перемещение осуществляется в плоскости xz, поэтому оно реализуется с помощью следующего кода.

```
cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);  
cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
```

Вращение камеры сводится к обновлению переменной `cam.dir.y`, которая затем передается в функцию, конструирующую матрицу камеры (эта функция вызывается в конце приведенного выше фрагмента кода). Интересной особенностью листинга, с которым мы ознакомились, является переменная `turning`. После запуска демонстрационной программы в поле зрения появляется копия танка, расположенная прямо перед камерой. Эта копия представляет танк, управляемый игроком. При нажатии клавиш со стрелкой вправо и стрелкой влево модель танка немного поворачивается относительно камеры (фактически происходит два поворота: один, когда танк вращается относительно камеры, и другой — когда танк вместе с камерой вращаются относительно игрового поля; в данном случае нас интересует поворот первого типа). Как только игрок отпускает клавишу, танк возвращается в исходное положение, разворачиваясь в том же направлении, куда направлена камера. Это реализуется с помощью переменной `turning`, отслеживающей состояние “повернутости” танка относительно камеры. Когда игрок хочет сделать поворот, он нажимает клавишу со стрелкой вправо или со стрелкой влево. Пока клавиша нажата, угол поворота танка относительно камеры возрастает до некоторого предельного значения (в данном случае это 15°), после чего поворот прекращается. Если же игрок отпускает клавишу, танк постепенно восстанавливает исходную ориентацию относительно камеры. Это очень популярный эффект, который используется во многих играх подобного типа, причем его реализация занимает всего несколько строк кода. Конечно же, в играх, в основе которых — реальная физическая модель, этот эффект закладывается изначально; ну, а для наших целей достаточно хорошо работает описанный выше прием.

## Генерация игрового поля

Игровое поле состоит из определенного количества статических объектов. Эти объекты являются копиями главных объектов и размещаются в фиксированных точках игрового поля. Для каждого объекта код выполняет перенос главного объекта, его преобразование и размещение в основном списке многоугольников, предназначенных для визуализации (по ходу дела предпринимается попытка выполнить отбраковку). Основное достоинство этого подхода в том, что исходные объекты при этом не изменяются. Поскольку используются только преобразованные вершины, начальные модели остаются целыми и невредимыми. Именно для такого случая и предусмотрено наличие двух массивов для хранения вершин. Приведем фрагмент кода демонстрационной программы, в котором в мировой системе координат размещаются модели башен.

```
// Размещаем башни в мировой системе координат  
for (index = 0; index < NUM_TOWERS; index++)  
{  
    // Сбрасываем параметры объекта (это относится только к  
    // флагам обратных поверхностей и удалению объектов)  
    Reset_OBJECT4DV1(&obj_tower);  
  
    // Задаем положение башни  
    obj_tower.world_pos.x = towers[index].x;  
    obj_tower.world_pos.y = towers[index].y;
```

```

obj_tower.world_pos.z = towers[index].z;

// Проверяем, не нужно ли отбраковать данный объект
if (!Cull_OBJECT4DV1(&obj_tower, &cam,
    CULL_OBJECT_XYZ_PLANES))
{
    // Если мы дошли до этого места, то объект с данными
    // мировыми координатами является видимым, и его
    // можно помещать в список визуализации. Выполняем
    // преобразование локальных координат модели в
    // мировые координаты
    Model_To_World_OBJECT4DV1(&obj_tower);

    // Помещаем объект в список визуализации
    Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list,
        &obj_tower);
} // if

} // for

```

Видите, как все просто? Ну и, наконец, рассмотрим, как реализуется модель, управляемая игроком.

### Визуализация модели танка игрока

Созданный нами игровой процессор не обладает большими функциональными возможностями. Хотя в нем имеется камера, ее нельзя разместить на объекте и сделать так, чтобы она перемешалась вместе с ним. Это нужно делать вручную. В рассматриваемой демонстрационной программе все операции используют известное положение камеры. Я провожу вектор направления камеры и сдвигаю модель танка вдоль него на некоторое расстояние (приблизительно 400 единиц), а затем — немного вниз относительно камеры. После этого я поворачиваю модель танка (эта модель немного отличается от остальных, разбросанных по всему игровому полю) исходя из заданного угла обзора, выполняю необходимые преобразования танка, размещаю его в мировой системе координат — и мы получаем танк игрока.

Код всего вышеописанного процесса довольно короткий.

```

// Размещение модели игрока в мировой системе координат

// Сброс параметров объекта (это относится только к флагам
// обратных поверхностей и удалению объектов)
Reset_OBJECT4DV1(&obj_player);

// Задаем положение танка
obj_player.world_pos.x = cam.pos.x+400*Fast_Sin(cam.dir.y);
obj_player.world_pos.y = cam.pos.y-50;
obj_player.world_pos.z = cam.pos.z+400*Fast_Cos(cam.dir.y);

// Генерируем матрицу поворота вокруг оси y.
Build_XYZ_Rotation_MATRIX4X4 (0, cam.dir.y+turning,
    0, &mrot);

// Поворачиваем локальные координаты объекта
Transform_OBJECT4DV1(&obj_player, &mrot,

```

```
TRANSFORM_LOCAL_TO_TRANS,1);
```

```
// Переходим в мировые координаты  
Model_To_World_OBJECT4DV1(&obj_player,  
    TRANSFORM_TRANS_ONLY);
```

```
// Заносим объект в список визуализации  
Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj_player);
```

Обратите внимание на строки, выделенные полужирным шрифтом. В них видно, как путем поворота модели танка относительно текущего направления камеры создается видимость того, что танк сворачивает с прямой дороги.

#### НА ЗАМЕТКУ

В большинстве демонстрационных программ используется 16-битовый графический режим. Однако горькая правда заключается в том, что при переходе к более серьезной графике с затенением и текстурой станет все труднее поддерживать нормальную частоту кадров для программно-ориентированной растеризации и 16-битовой графики. Поэтому, чтобы сохранить высокую скорость, нам придется поработать в 8-битовых режимах, а также изучить возможности оптимизации, связанные с применением палитры.

## Резюме

Похоже, я уже стер пальцы о клавиатуру, но глава стоила того. В ней можно многому научиться. Конечно же, вы можете осваивать только тот материал, который необходим для создания трехмерного игрового процессора, но лично я предпочитаю знать все до малейших деталей и уметь выполнять одни и те же операции различными способами. Надеюсь, что вы согласны со мной в этом.

В данной главе рассмотрено множество вопросов: о структурах данных для трехмерных объектов, загрузке объектов с диска, преобразовании локальных координат в мировые, преобразовании мировых координат в координаты камеры, в аксонометрические и в экранные координаты, отбраковке объектов, удалении обратных поверхностей, использовании четырехмерных координат... Вы узнали, почему далеко не всегда матрицы являются наилучшим способом выполнения преобразований координат, как выполняется визуализация объектов в каркасном режиме, и наконец, разобрались, как работает UVN-модель камеры. Кроме того, вы познакомились с массой демонстрационных программ. Теперь пришло время придать нашим трехмерным объектам большую реалистичность с помощью затенения и освещения. Кроме того, нам придется освоить трехмерное отсечение, поскольку откладывать реализацию этой операции уже просто невозможно. И конечно же, нам понадобится более совершенный загрузчик трехмерных моделей.

# ЧАСТЬ III

## Основы трехмерной визуализации

### В этой части...

#### Глава 8

Основы моделирования освещения и поверхностей тел 629

#### Глава 9

Интерполяционные методы затенения и аффинное отображение текстур 749

#### Глава 10

Отсечение в трехмерном пространстве 879

#### Глава 11

Организация буфера глубины и видимость 957



# ГЛАВА 5

## Основы моделирования освещения и поверхностей тел

### В этой главе...

• Основные модели освещения в компьютерной графике	630
• Освещение и растеризация треугольников	652
• Затенение в реальном мире	671
• Сортировка по глубине и алгоритм художника	714
• Работа с новыми форматами моделей	721
• Обзор программных инструментов для создания трехмерных моделей	744

В этой главе мы намерены **придать** реалистичность трехмерным изображениям путем их **визуализации** не в виде каркасов, а в виде сплошных геометрических тел. Следовательно, необходимо изучить принципы моделирования **освещения** и тени. Кроме того, концепция удаления невидимых обратных поверхностей теперь будет играть более важную роль, чем в случае каркасов, поскольку объект уже не является прозрачным. В данной главе обсуждаются общие **принципы** имитации освещения и тени как в 8-битовой, так и в 16-битовой версиях. Кроме того, мы собираемся усовершенствовать наш небольшой загрузчик файлов в форматах **PLG** и **PLX**, чтобы он **поддерживал** и другие форматы, и нам не пришлось самим вручную создавать модели. Ниже приведен список вопросов, **подлежащих** рассмотрению:

- основные модели освещения;
- источники общего света (т.е. свет, не имеющий определенного источника);
- источники рассеянного света;
- зеркальное отражение света;
- направленный свет;
- точечные источники;
- световые пятна;
- растеризация треугольников;
- битовая глубина освещения;
- однородное затенение<sup>1</sup>;
- плоское затенение;
- затенение по Гуро;
- затенение по Фонгу;
- упорядочение по глубине;
- модел и загрузки.

## Основные модели освещения в компьютерной графике

В компьютерной графике тема **освещения** составляет большой и важный раздел; для ее понимания требуется в той или иной мере изучить поведение фотонов и принципы их взаимодействия с веществом. Известно, что фотоны переносят энергию и обладают как волновыми, так и корпускулярными свойствами (т.е. свойствами **частиц**). Фотоны (как и частицы) могут обладать координатами и скоростью, а также (как и волны) частотой. Нарис. 8.1 представлена шкала частот электромагнитного излучения, а также указано, каким частотным интервалам отвечает **инфракрасный**, **видимый**, **ультрафиолетовый** свет, рентгеновские лучи и т.д. Известно, что свет любой частоты перемещается с одной и той же скоростью — скоростью света. Эта скорость обозначается как  $c$ , а ее величина приблизительно равна  $3.0 \times 10^8$  м/с. Соотношение между частотой света  $F$  (в герцах) и длиной волны  $X$  (в метрах) имеет следующий вид.

### Уравнение 8.1. Связь частоты и длины волны

$$F = c/\lambda .$$

Например, ярко-красный свет, длина волны которого приблизительно равна 700 nm (нанометр — это  $10^{-9}$  м), имеет частоту, равную  $F = 3.0 \cdot 10^8 / (700 \cdot 10^{-9}) \approx 4.29 \cdot 10^{14}$  Hz .

Можно подумать, что свет распространяется с чрезвычайно большой скоростью — однако если вам нужно перемещаться в пространстве на огромные **расстояния**, например, между галактиками, то оказывается, что эта скорость ужасно мала. Может быть, и можно перемещаться со скоростью, превышающей скорость света. Просто до этой ско-

<sup>1</sup> Затенение (shading) в данном случае не связано с **обработкой теней**, а означает процесс закрашивания тем или иным цветом (или цветами). — *Прим. перев.*

рости невозможно разогнаться, потому что для этого понадобится бесконечная энергия — но это уже совсем другая история...

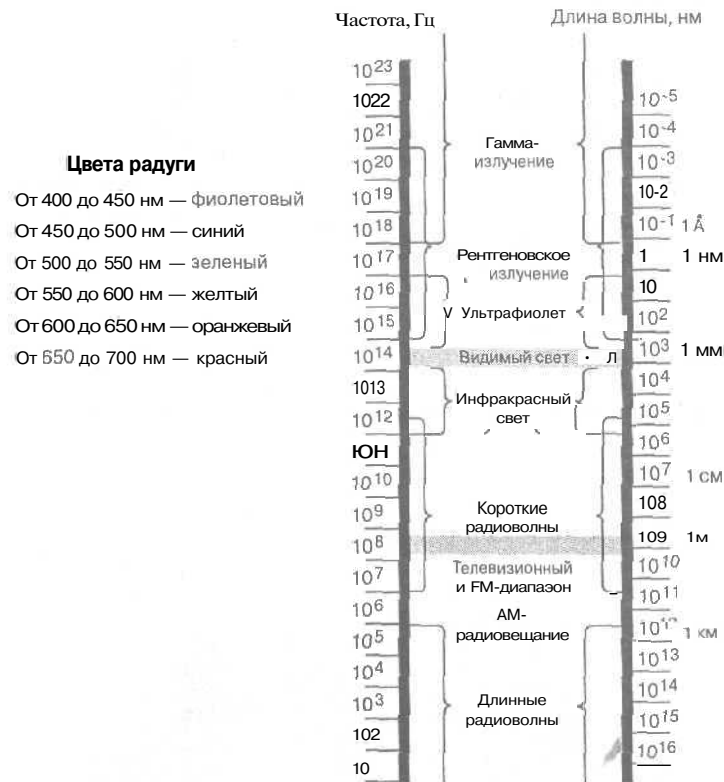


Рис. 8.1. Спектр электромагнитных волн

Цвет, воспринимаемый глазом при попадании в него электромагнитного излучения, непосредственно зависит от длины волны этого излучения (или, что то же самое, от его частоты). Это все хорошо, но чем определяется интенсивность света? Оказывается, интенсивность (или яркость) зависит от потока фотонов (т.е. от количества фотонов, прошедших через единичную поверхность за единицу времени). Таким образом, полная энергия, поглощаемая поверхностью, площадь которой равна  $S$ , определяется как суммарная энергия всех фотонов, падающих на эту поверхность за единицу времени. Следует учесть, что энергия, приходящаяся на один фотон, зависит от его частоты.

Конечно же, в основе большинства трехмерных графических процессоров не заложены концепции реальных фотонов или реального вещества, но вообще говоря, используемые процессорами модели базируются именно на этих концепциях. Почти во всех пакетах визуализации трехмерных изображений происходит отслеживание хода лучей (raytrace), суть которого заключается в том, чтобы проследить путь отдельных лучей (или фотонов) с точки зрения пользователя, который смотрит на экран. В ходе этого процесса определяется, как лучи падают на находящиеся в поле зрения предметы и, следовательно, каким будет цвет каждого пикселя экрана. Описанный выше процесс проиллюстрирован на рис. 8.2 на примере нескольких простых объектов. На этом рисунке показан результат визуализации методом бегущего луча (обратите внимание на тени и блики).

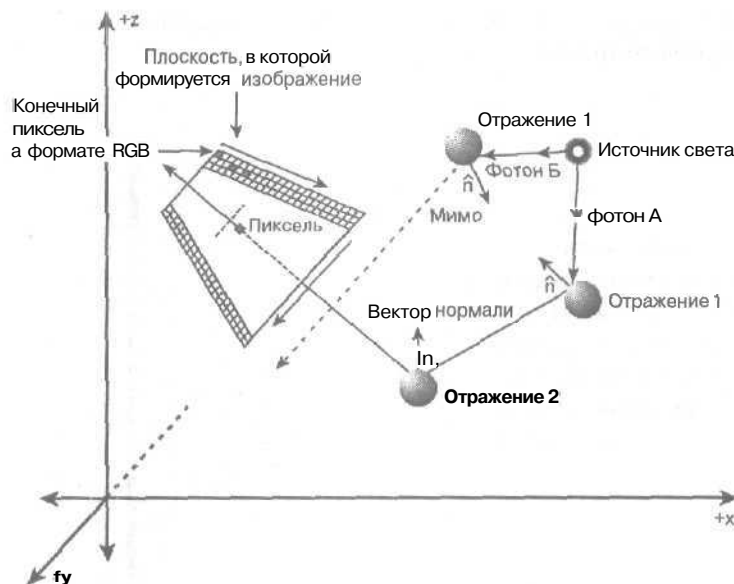


Рис. 8.2. Физика отслеживания лучей

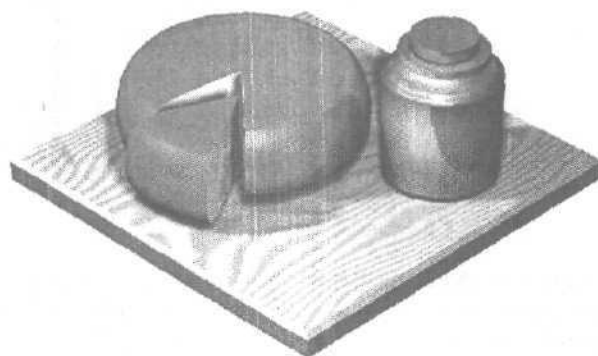


Рис. 8.3. Изображение предметов, полученное в результате отслеживания лучей

Кроме того, существуют такие широко известные методы, как *отображение фотонов* (photon mapping), *метод диффузного отражения* (radiosity) и другие, которые моделируют взаимодействие излучения с предметами, на которые оно падает, с помощью более сложных методов. Главная особенность этих методов заключается в том, чтобы вид предметов был максимально приближен к реальности.

Однако стремление максимально приблизиться к реальности ограничивается возможностями аппаратного и программного обеспечения, с которым мы работаем. В нашем случае создается программно-ориентированный процессор, поэтому он не может быть таким же сложным, как аппаратно-ориентированный. В этом и заключается данный этап работы — посмотреть, насколько совершенный программный процессор можно разработать и насколько реальное освещение он обеспечит. Если мы будем исходить из этого, то сможем дальше продвинуться в понимании моделей освещения, что влечет

ствии позволит получить максимальную отдачу от аппаратуры, применяя такие методы, как затенение пикселей и вершин. Итак, пора переходить к деталям.

## Цветовые модели и материалы

Материал, из которого состоит объект, определяет взаимодействие его поверхности с падающими фотонами. Пусть, например, белый свет (в состав которого входят частоты из всего видимого диапазона) падает на некоторую поверхность. Вообще говоря, эта поверхность обладает целым рядом свойств, определяющих ее взаимодействие со светом. Определенная часть света отражается от поверхности (именно отраженный свет определяет цвет поверхности), а оставшаяся часть — поглощается. Кроме того, объект может быть частично проницаемым для света. В этом случае часть электромагнитного излучения проходит сквозь объект (при этом свет, как правило, преломляется). Наконец, попадающие на материал фотоны могут вызвать его свечение (флуоресценцию или фосфоресценцию). Кроме того, все перечисленные процессы зависят от частоты и интенсивности падающего света. Таким образом, моделирование взаимодействия света с веществом — это сложная задача. Обычно материалы моделируются с помощью нескольких параметров, таких как отражательная способность, коэффициент рассеивания, коэффициент зеркального отражения, излучательная способность и т.д.

Перейдем к более конкретным концепциям. Как известно, цвета на экране компьютера представлены в формате RGB или 8-битовом режиме. В дальнейшем мы сосредоточим внимание на форматах RGB, в которых каждый канал представлен некоторым числом. Эти числа могут изменяться в диапазоне от 0 до 1 или, скажем, от 0 до 255. Например, в 24-битовом формате RGB на один канал приходится 8 битов, поэтому, например, чисто красный цвет представляется как (255,0,0).

Если же цвета представлены в формате с плавающей запятой, в котором интенсивность каждого цвета задается числом в диапазоне от 0 до 1, то чисто красный цвет может иметь вид (1,0,0). Оба представления — и (255,0,0), и (1,0,0) — задают один и тот же цвет. С форматом, в котором используются числа с плавающей запятой, легче производить математические операции. С другой стороны, формат, в котором задаются те числа, которые будут заноситься непосредственно в видеобuffer, обрабатывается быстрее, поскольку не требуется преобразование чисел с плавающей запятой в целые числа. Рассмотрим две операции, которые можно выполнять с цветами: сложение и модуляцию.

### Сложение цветов

Пусть в формате RGB задано два цвета:  $C_1(r_1, g_1, b_1)$  и  $C_2(r_2, g_2, b_2)$ . Тогда над этими цветами можно произвести операцию сложения:

$$C_{\text{sum}} = C_1 + C_2 = (r_1 + r_2, g_1 + g_2, b_1 + b_2).$$

Однако при этом возникает проблема, связанная с переполнением. Если результат сложения двух компонентов в каком-то канале превышает максимально допустимую величину, то результат, независимо от используемого формата, нуждается в переопределении. Таким образом, если используется 8-битовое представление (в котором интенсивность цвета задается величинами в интервале от 0 до 255), то при сложении цветов  $C_1$  и  $C_2$  понадобится определить результат сложения в каждом канале следующим образом:

$$C_{\text{sum}} = C_1 + C_2 = (\min(r_1 + r_2, 255), \min(g_1 + g_2, 255), \min(b_1 + b_2, 255)),$$

где  $\min(x, y)$  — минимальное из значений  $x$  и  $y$ .

Этот прием помогает избежать переполнения — суммарная величина ни в каком канале не превысит значения 255. В противном случае могло бы возникнуть переполнение, что привело бы к абсолютно неверному результату.

## Модуляция цветов

Еще один интересный процесс — модуляция цветов, которая, по сути, соответствует умножению. Рассмотрим простой пример, в котором наш первый цвет  $C_1$  умножается на скаляр  $s$ :

$$S_{\text{modulated}} = s \cdot C_1 = (s \cdot r_1, s \cdot g_1, s \cdot b_1)$$

В данном случае  $s$  может принимать любое значение от нуля до бесконечности. Если оно равно 1.0, то цвет остается неизменным, а если 2.0, то его яркость возрастает в два раза. Если значение  $s$  равно 0.1, то получаем цвет, яркость которого в 10 раз меньше яркости исходного цвета и т.д. Однако нам снова нужно соблюдать осторожность и следить за тем, чтобы результат не превысил максимально допустимого значения. В качестве примера такого однородного изменения цвета ознакомьтесь с демонстрационным файлом DEMO118\_1.CPP[EXE]. Яркость изображения изменяется путем нажатия клавиш со стрелкой вверх и стрелкой вниз. Обратите внимание, что если величина яркости начинает превышать некоторое предельное значение, наступает насыщение RGB-каналов и текстура искажается. Это можно исправить. Если в процессе тестирования модуляции значение в каком-либо из каналов достигло максимально допустимой величины (например 255), то нужно сделать так, чтобы цвет пикселя больше не менялся. Увеличение значений интенсивности других каналов для данного пикселя приведет к изменению его цвета, поскольку нарушится соотношение между интенсивностями различных каналов.

Рассмотренный выше вид модуляции приводит к обычному изменению яркости. Однако возможна модуляция другого вида, при которой интенсивности соответствующих каналов перемножаются. Ниже приведен результат модуляции цветов  $C_1$  и  $C_2$ :

$$C_{\text{modulated}} = C_1 \cdot C_2 = (r_1 \cdot r_2, g_1 \cdot g_2, b_1 \cdot b_2)$$

Заметим, что этот вид модуляции несколько сложнее, чем предыдущий. Нужно выработать соглашения, определяющие, что следует понимать под умножением. Например, если формат RGB цвета  $C_1$  имеет вид (0..1, 0..1, 0..1), то независимо от формата цвета  $C_2$  переполнение не возникнет, и получается, что интенсивность цвета  $C_2$  модулируется интенсивностью цвета  $C_1$ . В этом и заключается суть отображения цветов. Пусть у нас есть текстура освещения LIGHT\_TEXTURE[x][y] и текстура цветов COLOR\_TEXTURE[x][y]. Если для каждого пикселя текстуры LIGHT\_TEXTURE[x][y] величины интенсивностей каналов принимают значения от 0 до 1, то эти значения можно умножить на соответствующие интенсивности каналов в текстуре COLOR\_TEXTURE[x][y]. Результат этой операции представлен на рис. 8.4 вместе с исходными текстурами LIGHT\_TEXTURE[x][y] (обратите внимание, что она монохроматическая) и COLOR\_TEXTURE[x][y]. Единственная проблема, возникающая в связи с рассматриваемой модуляцией, заключается в том, что иногда результат получается неожиданным, и может потребоваться более сложная последовательность модуляций (см. уравнения, представленные ниже).

### Уравнение 8.2а. Затенение пикселей с аддитивным компонентом и модуляцией

$$\begin{aligned} \text{Pixel\_dest}[x, y] = & \text{pixel\_source}[x, y]_{\text{a}} \cdot \text{ambient} + \\ & + \text{pixel\_source}[x, y]_{\text{rgb}} \cdot \text{light\_map}[x, y]_{\text{rgb}}. \end{aligned}$$

**Уравнение 8.2б. Затенение пикселей с простой аддитивной модуляцией (лучше и быстрее)**

$$\text{Pixel\_dest}[x, y] = \text{pixel\_source}[x, y]_{\text{rgb}} + \text{ambient} \cdot \text{light\_map}[x, y]_{\text{lc}}$$

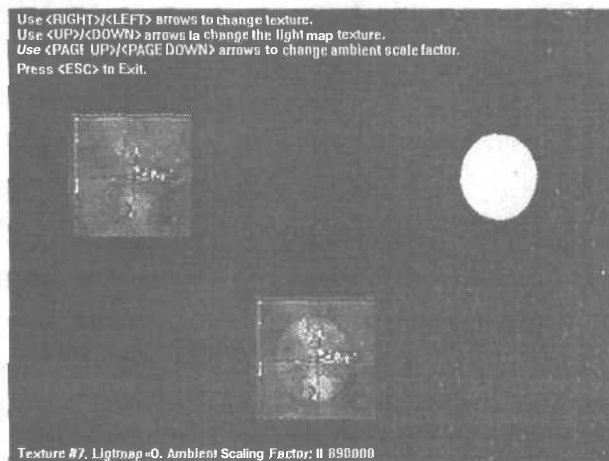


Рис. 8.4. Световая модуляция в действии

**НА ЗАМЕТКУ**

Запись  $[x, y]$  означает, что операция выполняется для всех точек изображения. Нижний индекс *rgb* означает, что действия выполняются отдельно для каждого канала — красного, зеленого и синего.

Суть первой из приведенных выше формул заключается в том, что для **каждого** пикселя текстуры берется определенное процентное отношение интенсивности, после чего к полученному результату прибавляется результат **модуляции** данной текстуры **текстурой** освещения. Этот процесс проиллюстрирован на рис. 8.5. Нетрудно убедиться, что результат несколько ярче, чем обычная модуляция. По **сути**, мы не только выполняем модуляцию исходной поверхности, но и добавляем ее в "смягченном" виде. Переменная *ambient* — это обычный скаляр, **величина** которого определяет, какая часть исходной текстуры добавляется к конечному результату, который выводится на экран. Заметим, что на практике результат, полученный по формуле 8.2б, выглядит намного лучше. Суть этой формулы в том, что световая текстура, умноженная на некоторый множитель, прибавляется к исходной.

В качестве примера использования наложения света и модуляции ознакомьтесь с демонстрационными программами `ДЕМОП8_2.CPP|EXE` (в ней используется первая формула) и `ДЕМОП8_2б.CPP|EXE` (в ней используется вторая формула). Эти примеры **позволяют** модулировать текстуру поверхности некоторой световой текстурой с помощью формул 8.2а и 8.2б. Обратите внимание на различия работы обеих демонстрационных программ. Во второй из них используется сложение цветов, в результате **чего** получается зеленый цвет, а в **первой** — модуляция. Оба рассмотренных метода могут оказаться полезными. Все зависит от того, чего вы добиваетесь — реалистичности или презентабельного внешнего вида. Для управления изображением в обеих демонстрационных **программах** применяются клавиши со стрелками.

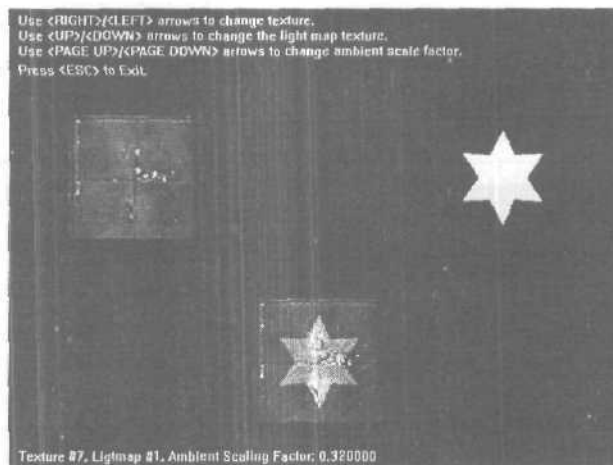


Рис. 8.5. Аддитивная и мультипликативная модуляция

Перейдем к рассмотрению задач, возникающих в процессе моделирования освещения. Даже в простом случае, когда у нас имеется одна поверхность с рисунком, на которую проецируется некоторое освещение (далее этот процесс рассматривается намного подробнее), приходится выполнять ряд умножений, сложений, а также применять условные операторы, чтобы проверить, не возникло ли переполнение, а в случае его возникновения — предпринимать соответствующие меры. Таким образом, необходимо позаботиться о том, чтобы используемая числовая модель была самой лучшей и чтобы в ходе получения результата в этой модели выполнялось как можно меньше преобразований и промежуточных действий. Кроме того, если применяется 8-битовый режим, то следует учесть, что он использует палитру, а при этом приходится прибегать к еще более сложным трюкам. С другой стороны, в 8-битовых режимах многие процессы идут быстрее, поскольку в них можно применять таблицы соответствия.

### Альфа-смешивание

Последняя операция, о которой хотелось бы рассказать, — альфа-смешивание. В ней сочетаются концепции сложения и модуляции. *Альфа-смешивание* (alpha blending) — это взвешенное наложение одного или нескольких смешиваемых цветов. Смешивание проявляется в том, что каждый из исходных цветов берется в определенном соотношении, после чего все смешиваемые цвета складываются. На рис. 8.6 приведены два исходных битовых образа, а также результат их альфа-смешивания. Альфа-смешивание применяется для имитации прозрачности, тени и других подобных эффектов. Математическое выражение этой операции выглядит довольно просто, а его суть заключается в том, что два (или несколько) исходных изображений сопрягаются с помощью определенного множителя (как правило, он обозначается буквой *альфа*, а соответствующая переменная в коде носит имя alpha), в результате чего полученный цвет представляет собой сумму взвешенных исходных цветов. Приведем формулу альфа-смешивания для двух источников.

#### Уравнение S.3. Альфа-смешивание двух источников

$$\text{Pixel\_dest}[x, y] = \alpha \cdot \text{pixel\_source1}[x, y]_b + \\ + (1 - \alpha) \cdot \text{pixel\_source2}[x, y]_{fc}.$$

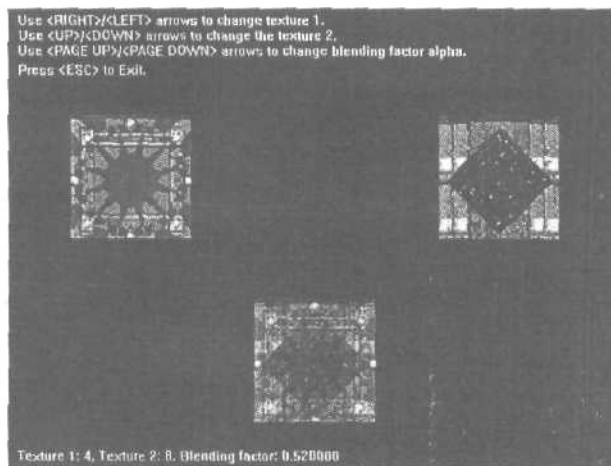


Рис. 8.6. Альфа-смешивание в действии

Таким образом, если  $\alpha = 1$ , то пиксели получившегося в результате изображения будут совпадать с пикселями первого изображения. Если же  $\alpha = 0.9$ , то результирующий цвет будет представлять собой смесь, включающую 90% первого изображения и 10% второго. Несмотря на то, что представленный алгоритм довольно простой, не следует забывать, что данный процесс необходимо выполнить для всех пикселей, в каждом из которых имеется по три канала. Альфа-смешивание демонстрируется в программе DEMOII8\_3.CPP\EXE. С помощью клавиш со стрелками вправо и влево можно изменять значение переменной alpha от 0 до 1 и наблюдать, что получится в результате.

Теперь, когда мы немного ознакомились с тем, какие операции можно выполнять с цветами, поговорим о типах освещения.

## Общий свет

*Общий свет* (ambient light) — это такой свет, который окружает нас и не имеет определенного источника. Такое освещение является результатом испускания (отражения и рассеяния) света всеми объектами, что приводит к повышению уровня освещенности. Общее освещение моделируется путем прибавления к исходной интенсивности постоянного слагаемого, величина которого является результатом произведения исходной интенсивности общего света  $I_{\text{ambient}}$  на общую отражательную способность поверхности  $C_{\text{ambient}}$ . (Авторы многих книг используют для обозначения отражательной способности букву  $k$ , однако при моделировании цвета удобнее подразумевать и под отражательной способностью поверхности, и под светом именно цвет, поэтому мы будем обозначать отражательную способность буквой  $C$ ). Это слагаемое может выражаться в формате RGB или быть скаляром, представляющим монохроматический свет. Отличительной особенностью такого освещения является то, что яркость всех находящихся в поле зрения **многоугольников** возрастает независимо от того, где находится источник света. Таким образом, интенсивность света, падающего на поверхность  $S$  в некоторой точке, при рассеянном освещении равна

$$I_{\text{total}} = C_{\text{ambient}} \cdot I_{\text{ambient}}$$

## Рассеянный свет

*Рассеянный свет* (diffuse light) — это свет, который рассеивается поверхностью объекта, что является результатом **шероховатости** этой поверхности. Рассеяние происходит равномерно по всем направлениям и не зависит от положения наблюдателя и рассеивающего

предмета. На рис. 8.7 изображен луч света, падающий на поверхность. Поверхность рассеивает свет во всех направлениях, но интенсивность рассеяния зависит от угла, под которым свет попадает на поверхность из источника. Результирующая интенсивность пропорциональна проекции нормали к поверхности на направление хода световых лучей.

НА ЗАМЕТКУ

Чтобы облегчить математические вычисления, предположим, что отражательная способность выражается в формате RGB числами с плавающей точкой (0..1, 0..1, 0..1), а интенсивность — скаляром, величина которого выражается числом с плавающей точкой. Кроме того, обратите внимание на нижний индекс, выделенный полужирным шрифтом. Он представляет тип освещения, для которого вычисляется конечная интенсивность (в данном случае *a* обозначает ambient — общий).

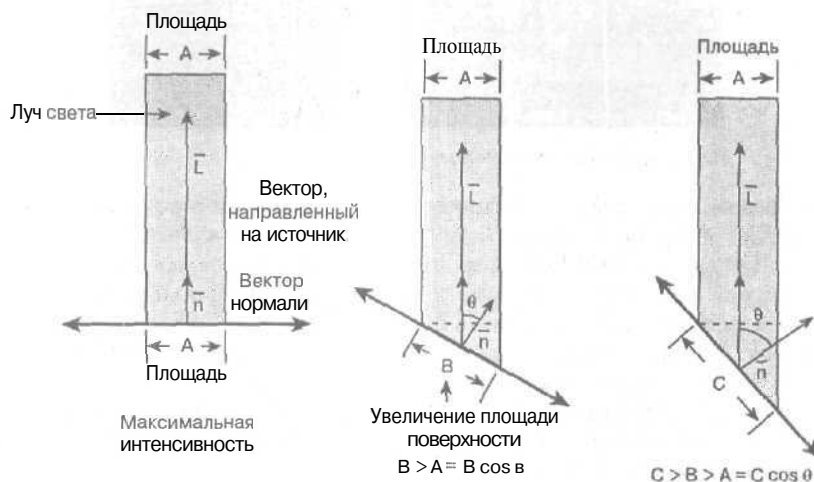


Рис. 8.7. Геометрия диффузного рассеяния

#### Уравнение 8.4. Интенсивность диффузного освещения в зависимости от угла между нормалью к поверхности и направлением освещения

$I = (\mathbf{n} \cdot \mathbf{l}) = \cos \theta$ . Примечание:  $\mathbf{n}$  и  $\mathbf{l}$  должны быть единичными векторами.

Справедливость сказанного выше подтверждается тем, что при увеличении угла  $\theta$  от  $0^\circ$  до  $90^\circ$  площадь поверхности уменьшается (она становится параллельна лучам, идущим от источника) и величина энергии, отражаемой поверхностью единичной площади, уменьшается. И наоборот, по мере того, как положение поверхности становится перпендикулярным направлению освещения (угол  $\theta$  меняется от  $90^\circ$  до  $0^\circ$ ), ее площадь возрастает, количество энергии, приходящееся на единицу поверхности, увеличивается и свет рассеивается сильнее. Максимум достигается в тот момент, когда вектор нормали  $\mathbf{n}$  и вектор, направленный на источник, становятся параллельными.

Диффузное отражение еще называют *отражением Ламберта* (Lambertian reflection), Теория отражения строится на концепции, что различные поверхности изготовлены из материалов, по-разному взаимодействующих со светом. Один из возможных видов взаимодействия — рассеивание определенной компоненты света по всем направлениям. Это происходит из-за того, что поверхность состоит из множества микроскопических участков, вектор нормали к которым направлен случайным образом. В действительности в основе этого явления лежат более сложные физические процессы. Интенсивность и угол

рассеянного света может зависеть от его частоты. Однако в нашей модели предполагается, что поверхность объекта, изготовленного из того или иного материала, характеризуется **цветовым коэффициентом** диффузного отражения  $Rs_{\text{diffuse}}$ , которым определяется интенсивность света, диффузно отраженного от поверхности данного материала. Отметим, что этот коэффициент представляет собой не скаляр, а **цвет** в формате RGB, поэтому может оказаться, что поверхность отражает только свет определенной частоты (например, синий цвет, если  $Rs_{\text{diffuse}} = (0, 0, 1)$ ). Другой пример — поверхность, которая в равной мере диффузно отражает все цвета и для которой отсутствуют другие виды рассеяния, такие как зеркальное. Для такой поверхности  $Rs_{\text{diffuse}} = (1, 1, 1)$ . Математическое выражение модели диффузного освещения с одним источником имеет **следующий** вид.

#### Уравнение 8.5. Диффузное освещение с одним источником

$$I_{\text{totald}} = Rs_{\text{diffuse}} \cdot I_{\text{diffuse}} \cdot (n \cdot l).$$

С учетом только что рассмотренного диффузного слагаемого модель освещения, и которой имеется только один источник с интенсивностью  $I_{\text{diffuse}}$ , принимает следующий вид.

#### Уравнение 8.6. Модель освещения одним источником света

$$I_{\text{totald}} = Rs_{\text{ambient}} \cdot I_{\text{ambient}} + (Rs_{\text{diffuse}} \cdot I_{\text{diffuse}} \cdot (n \cdot l)).$$

Чтобы учесть наличие нескольких источников освещения, достаточно просуммировать все диффузные слагаемые, а слагаемое, соответствующее общей освещенности, оставить неизменным.

#### Уравнение 8.7. Модель освещения с несколькими источниками света

$$I_{\text{totald}} = Rs_{\text{ambient}} \cdot I_{\text{ambient}} + Rs_{\text{diffuse}} \sum_{i=1}^n (I(i)_{\text{diffuse}} \cdot (n_i \cdot l_i)).$$

##### НА ЗАМЕТКУ

Возможно, вы обратили внимание, что в формулах имеется определенная избыточность. Например, нет необходимости использовать интенсивности и коэффициенты отражения (особенно в слагаемых, которые отвечают общему освещению). Можно просто перемножить эти величины, благодаря чему количество используемых констант уменьшится. Однако конструкция в уравнении 8.7 построена не оптимально, а с целью показать все независимые компоненты.

### Зеркальное отражение

Еще одно свойство света, которое мы наблюдаем, глядя на объекты, — зеркальное отражение источников света. Зеркальное отражение в основном обусловлено наличием большого количества микроскопических элементов поверхности, имеющих **одинаковое** направление. Модели зеркального освещения пытаются имитировать этот эффект. При этом важно учитывать положение источника света и наблюдателя, а также направление вектора нормали к поверхности. (Напомним, что при рассеянном освещении положение наблюдателя не играет роли.)

На рис. 8.8 схематически изображен **процесс** зеркального отражения. На этой схеме представлена поверхность S с вектором нормали n, единичный вектор v, направленный к наблюдателю, а также вектор r, **указывающий** направление отраженного луча (он образует с вектором нормали тот же угол, что и вектор l, направленный на источник света). В качестве эксперимента можно сделать следующее: возьмите предмет с зеркальной поверхностью

(например, гладкую пластмассовую пластинку) и, поместив ее перед собой, медленно поворачивайте и перемещайте до тех пор, пока не увидите отраженный в этой пластинке источник света, который находится в комнате. Это и есть зеркальное отражение.

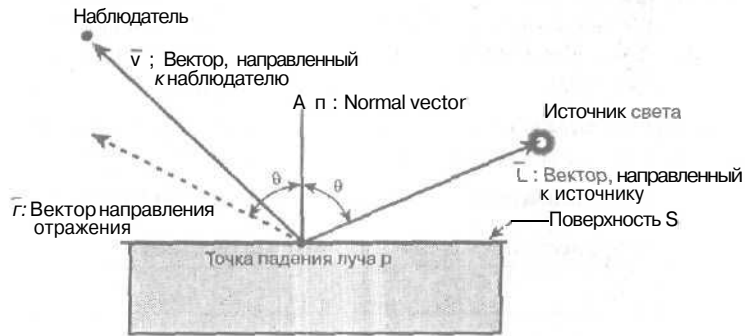


Рис. 8.8. Учет зеркального отражения

Изображение источника света, сформированное отраженными от поверхности предмета лучами, можно будет увидеть при условии, что векторы  $\mathbf{r}$ ,  $\mathbf{n}$  и  $\mathbf{v}$  находятся в одной плоскости, потому что поверхность предмета выступает в роли зеркала. Чем выше коэффициент зеркального отражения объекта, тем четче получается изображение источника света. По мере того, как угол между векторами  $\mathbf{r}$  и  $\mathbf{v}$  уменьшается до нуля, интенсивность отраженного света резко возрастает; с увеличением этого угла интенсивность зеркального отражения быстро уменьшается.

**СОВЕТ**

Обычно описанную выше модель называют моделью освещения по Фонгу (Phong illumination model), хотя чаще встречается название затенение по Фонгу (Phong shading), которое представляет собой метод, использующий интерполяцию нормалей в процессе затенения. Несмотря на то, что эти две концепции определенным образом связаны между собой, их не следует путать.

Чтобы смоделировать рассмотренный выше процесс, следует учесть направление векторов  $\mathbf{v}$  и  $\mathbf{r}$ , а также константу, характеризующую зеркальность поверхности моделируемого материала, т.е. ее коэффициент зеркального отражения. Здравый смысл подсказывает, что по мере уменьшения угла между векторами  $\mathbf{v}$  и  $\mathbf{r}$  интенсивность зеркального отражения должна возрастать. Если угол между вектором нормали к поверхности  $\mathbf{n}$  и вектором, направленным на источник, становится тупым, то источник больше не сможет освещать поверхность.

Как и в предыдущих случаях, нам понадобится численная характеристика поверхности, в роли которой теперь выступает коэффициент зеркального отражения. Обозначим его через  $R_{s\_specular}$ . Кроме того, для нашей модели нужна интенсивность источника  $I_{specular}$  и еще одна величина, определяющая концентрацию освещения, которую иногда называют степенью зеркального отражения (specular power или specular exponent). Обозначим последнюю переменную через  $sp$ . Приведем формулу, моделирующую процесс зеркального отражения света, падающего на поверхность из одного источника.

**Уравнение 8.8. Формула отраженного света для одного источника**

$$I_{total} = R_{s\_specular} \cdot I_{specular} \cdot \max(\mathbf{r} \cdot \mathbf{v}, 0)^{sp}.$$

Обратите внимание на наличие в этой формуле функции  $\max$ , благодаря которой результирующая интенсивность никогда не будет отрицательной.

В приведенном выше уравнении использовано скалярное произведение векторов. Можно было бы расшифровать его в соответствии с определением скалярного произведения и  $\mathbf{r} \cdot \mathbf{v} = |\mathbf{r}| |\mathbf{v}| \cos \alpha$ , где  $\alpha$  — угол между векторами  $\mathbf{r}$  и  $\mathbf{v}$ . Тогда формулу 8.8 можно переписать следующим образом:  $R_{\text{specular}} \cdot I_{\text{specular}} \cdot \cos^{\text{sp}} \alpha$ . При этом, конечно же, предполагается, что  $\mathbf{r}$  и  $\mathbf{v}$  — единичные векторы.

Как уже было сказано, угол между вектором  $\mathbf{l}$ , направленным на источник света, и вектором нормали к поверхности  $\mathbf{n}$  не должен превышать  $90^\circ$ , чтобы выполнялось соотношение  $\mathbf{n} \cdot \mathbf{l} > 0$ . Таким образом, при желании в уравнение можно добавить логический оператор  $(\mathbf{n} \cdot \mathbf{l}) > 0 ? 1 : 0$ , чтобы отключить вычисление зеркального отражения для тех точек поверхности  $S$ , для которых источник света находится позади этой поверхности. В результате приходим к следующей формуле.

**Уравнение 8.9. Формула отраженного света для одного источника с проверкой его доступности**

$$I_{\text{total}} = R_{\text{specular}} \cdot I_{\text{specular}} \cdot \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\text{sp}} \cdot ((\mathbf{n} \cdot \mathbf{l}) > 0 ? 1 : 0).$$

Рассмотрим, как нужно модифицировать эту формулу, если у нас не один источник, а несколько. Для этого достаточно просуммировать вклад каждого источника.

**Уравнение 8.10. Отраженный свет нескольких источников**

$$I_{\text{total}} = R_{\text{specular}} \cdot \sum_{i=1}^n (I(i)_{\text{specular}} \cdot \max(\mathbf{r}_i \cdot \mathbf{v}_i, 0)^{\text{sp}} \cdot ((\mathbf{n}_i \cdot \mathbf{l}_i) > 0 ? 1 : 0)).$$

Объединим в рамках одной модели слагаемые, отвечающие общему, рассеянному и зеркальному освещению:

$$\begin{aligned} I_{\text{totalads}} &= I_{\text{totala}} + I_{\text{totald}} + I_{\text{total}} = \\ &= R_{\text{ambient}} \cdot I_{\text{ambient}} + R_{\text{diffuse}} \cdot \sum_{i=1}^n I(i)_{\text{diffuse}} \cdot (\mathbf{n}_i \cdot \mathbf{l}_i) + \\ &+ R_{\text{specular}} \cdot \sum_{i=1}^n I(i)_{\text{specular}} \cdot \max(\mathbf{r}_i \cdot \mathbf{v}_i, 0)^{\text{sp}} \cdot ((\mathbf{n}_i \cdot \mathbf{l}_i) > 0 ? 1 : 0). \end{aligned}$$

**Светящиеся поверхности**

Смоделировать свечение легче всего. Создавая светящуюся поверхность, нужно задать интенсивность ее излучения (самопроизвольного свечения). Если бы это освещение было реальным, то его следовало бы добавить к остальным источникам света. Свет от этих источников диффузно и зеркально отражается от разных объектов. Однако этими процессами можно пренебречь, ограничившись только учетом собственной светимости самой поверхности.

Например, предположим, что в игровой сцене присутствует такой объект, как маяк. Пусть он изготовлен из материала с малыми коэффициентами общего и диффузного отражения. Таким образом, если маяк выключен, то видны только его общие контуры (благодаря окружающему освещению). Если же маяк включен, он засветится. На самом деле, это вовсе не свет, потому что в нашей модели маяк не излучает реальный свет, взаимодействующий с другими объектами, — он только выглядит светящимся. Описанный процесс проиллюстрирован на рис. 8.9. На нем изображен маяк и объект возле него, однако даже после включения маяка этот объект маяком не освещается (понятно, что в реальной жизни все не так). Реализация этого процесса тривиальна (см. нижеприведенное уравнение).

### Уравнение 8.11. Излучение

$$I_{\text{total}} = R s_{\text{emission}}.$$

Итак, цвет пикселя определяется только его собственной светимостью  $R s_{\text{emission}}$ , но никакие характеристики источника света в выражении не участвуют. Таким образом, общее уравнение, которым определяется модель освещенности, имеет вид.

### Уравнение 8.12. Общее уравнение освещенности

$$I_{\text{total}} = R s_{\text{ambient}} \cdot I_{\text{ambient}} + R s_{\text{emission}} + \\ + R s_{\text{diffuse}} \cdot \sum_{i=1}^n I(i)_{\text{diffuse}} \cdot (n_i \cdot l_i) + \\ + R s_{\text{specular}} \cdot \sum_{i=1}^n I(i)_{\text{specular}} \cdot \max(r_i \cdot v_i, 0)^{sp} \cdot ((n_i \cdot l_i) > 0 ? 1 : 0).$$

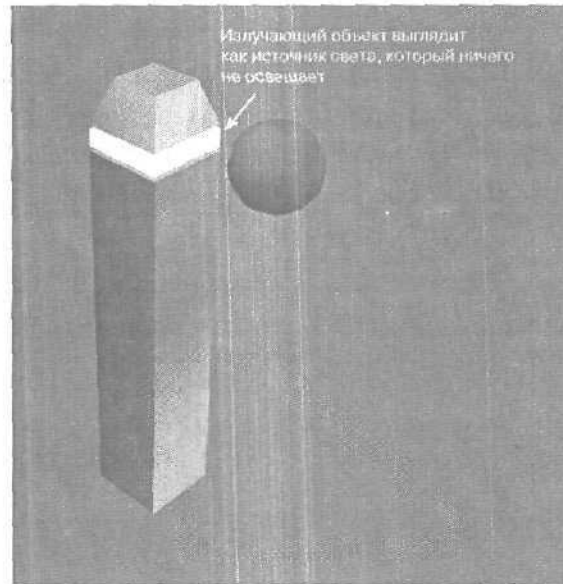


Рис. 8.9. Моделирование светящейся поверхности, не являющейся источником света

Последнее замечание: интенсивности  $I_{\text{ambient}}$ ,  $I_{\text{diffuse}}$  и  $I_{\text{specular}}$  могут как совпадать, так и различаться. Это соответствует тому, что в модели освещения одни источники воздействуют на объекты, отражающие диффузный свет, другие — на те объекты, которые отражают зеркальный свет и т.д. В нашем случае все интенсивности выбраны одинаковыми. Во многих случаях интенсивности всех  $i$ -тых компонент  $I_i$ , используемых во всех уравнениях, можно считать одинаковыми. В этом случае уравнения упрощаются и выглядят следующим образом:

$$I_{\text{total}} = R s_{\text{ambient}} \cdot I_{\text{ambient}} + R s_{\text{emission}} + \\ + R s_{\text{diffuse}} \cdot \sum_{i=1}^n I(i) \cdot (n_i \cdot l_i) + \\ + R s_{\text{specular}} \cdot \sum_{i=1}^n I(i) \cdot \max(r_i \cdot v_i, 0)^{sp} \cdot ((n_i \cdot l_i) > 0 ? 1 : 0).$$

Выглядит ужасно, не так ли? И все эти действия нужно выполнить для каждого пикселя и для всех формирующих изображение цветов. Работы очень много, а ведь мы даже не обсуждали виды **освещения**! Мы всего лишь рассмотрели процессы отражения **света** от поверхностей тел, но не сами источники света. Итак, поговорим об освещении и его интенсивности — еще один кошмар...

## Виды освещения

Конечный результат может оказаться вполне приемлемым, а **изображение** — выглядеть замечательно, несмотря на то, что **используется** предельно упрощенная модель **освещения**, а объекты **составляются** в виде многоугольников, образованных сеткой с крупными ячейками. С другой стороны, следует помнить о том, что при моделировании освещения операции выполняются с каждым пикселем, и даже если **освещение** имитируется с помощью грубой модели, то для получения правильного цвета и **интенсивности** пикселя над ним придется **выполнять** множество действий.

Еще одна тема, которую следует рассмотреть, — само освещение. Возможно, вы представляете себе источники света как небольшие штучки грушевидной формы, расположенные в спальне и офисе, однако их математическая модель несколько иная. В большинстве случаев в компьютерных моделях используются **источники** света трех видов:

- направленные;
- точечные;
- световые пятна.

Как ни странно, математические выражения, описывающие каждый из перечисленных выше источников света (скоро мы рассмотрим их подробнее), имеет весьма **отдаленное** отношение к реальной физике. В этом и заключается их прелесть.

## Направленное освещение

Направленное (directional) **освещение** — это такое освещение, положение источника которого не задается, поскольку он находится очень далеко (практически в бесконечности). При этом лучи света от такого источника, падающие на поверхность, можно **считать** параллельными (рис. 8.10). Направленное освещение обладает тем **свойством**, что его интенсивность не убывает с расстоянием от источника (поскольку он находится на **бесконечно большом расстоянии**). Направленное **освещение** характеризуется интенсивностью и цветом, и этих двух **показателей** достаточно для того, чтобы полностью задать такое **освещение**. Заметим, что нужно аккуратно выбирать обозначения. Напомним, что у нас уже есть несколько обозначений для коэффициентов отражения (общего, диффузного и зеркального), характеризующих отражательные свойства поверхности и **определяющих** модель освещения. Напомним также, что эти величины относились к тому материалу, из которого изготовлен объект; сейчас же речь пойдет о цвете самого освещения! Чтобы избежать путаницы, обозначим цвет освещения через  $C_{xxx}$ , где xxx — тип освещения. а его начальную **интенсивность** — через  $I0_{xxx}$ . Тогда для направленного освещения интенсивность всегда выглядит **следующим образом**.

### Уравнение 8.13. Интенсивность направленного освещения

$$I(d)_{dir} = I0_{dir} \cdot C_{dir}.$$

Обратите внимание, что интенсивность такого освещения не зависит от **расстояния** до источника, поэтому расстояние в формуле отсутствует.

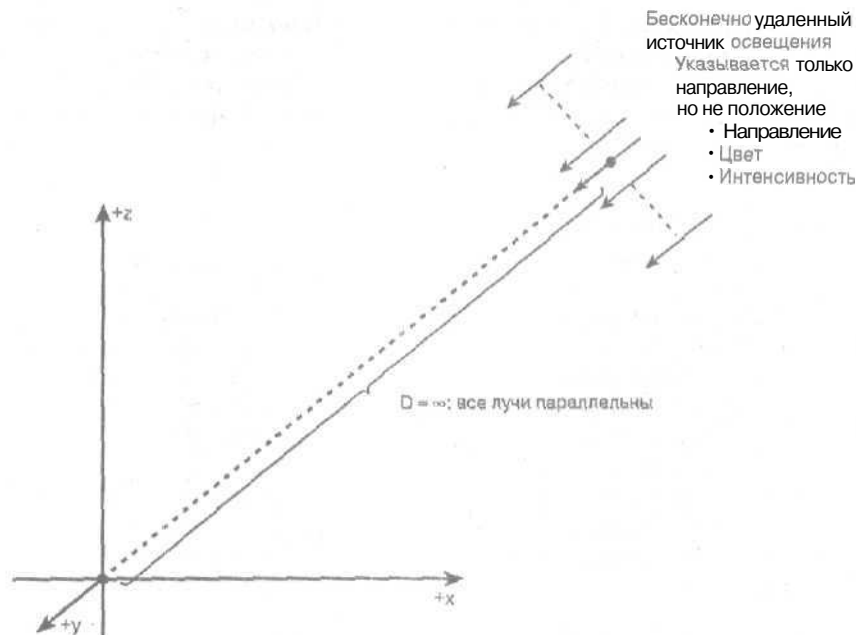


Рис. 8.10. Свойства бесконечно удаленных источников света

### Точечные источники

*Точечный источник света* (point light) моделируется светящейся точкой, расположенной в определенном месте трехмерного пространства (рис. 8.11). Модель точечного источника должна воспроизводить характеристики реального точечного источника, поэтому в ней следует предусмотреть убывание интенсивности **освещения** с увеличением расстояния от источника до **освещаемой** поверхности  $S$ . Такое ослабление, или *затухание* (attenuation) обычно **воспроизводится** с помощью трех констант: постоянного коэффициента затухания  $k_c$ , линейного коэффициента затухания  $k_l$  и квадратичного коэффициента затухания  $k_q$ . С учетом коэффициентов затухания и того, что расстояние от источника света, расположенного в точке  $p$ , до точки  $s$  поверхности  $S$  обозначается как  $d$ , интенсивность в точке  $s$  выражается следующим соотношением.

#### Уравнение 8.14. Интенсивность точечного источника света

$$I(d)_{\text{point}} = \frac{I_{0\text{ point}} \cdot C_{\text{point}}}{k_c + k_l \cdot d + k_q \cdot d^2}, \text{ где } d = |p - s|.$$

Чтобы понять, как выглядит эта функция, проведем с ее помощью некоторые вычисления. Не следует забывать о том, что все уравнения, входящие в модель освещения, всегда должны выражаться в терминах формата RGB. Это означает, что каждое **уравнение** — это на самом деле три уравнения: для красного, зеленого и синего **каналов**. Поскольку все эти уравнения имеют один и тот же вид, мы ограничиваемся тем, что записываем только одно уравнение, подразумевая, что оно **применимо** для всех трех каналов. Таким **образом**, не теряя общности (люблю это **выражение**), можно выполнить **все** математические вычисления для одного канала (другими словами, для монохроматической системы) и посмотреть, как будут выглядеть графики зависимостей различных физиче-

ских величин. Это позволит получить представление о том, каким будет отклик полноцветной RGB-системы. Приступим к решению поставленной задачи. Во-первых, примем, что  $I_{0\text{ point}} \approx 1.0$  и  $C_{I\text{ point}} = 1.0$ , поэтому их произведение равно единице, в силу чего наши результаты будут определенным образом нормированными. Проведем вычисления для трех разных наборов коэффициентов затухания:

набор 1:  $k_c = 1, k_l = 0, k_q = 0$ ;

набор 2:  $k_c = 0, k_l = 1, k_q = 0$ ;

набор 3:  $k_c = 0, k_l = 0, k_q = 1$ .

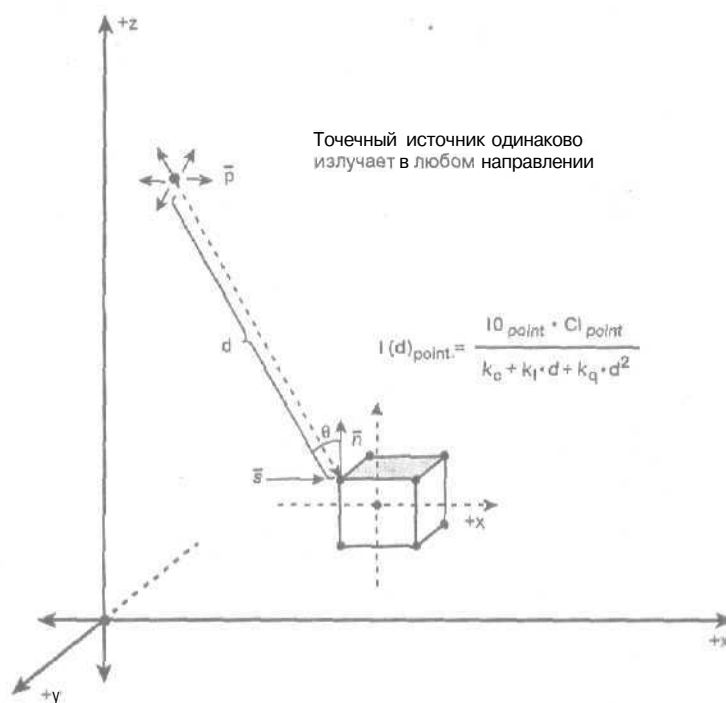


Рис. 8.11. Свойства точечного источника

Полученные кривые приведены на рис. 8.12. Как видно из графиков, интенсивность как для первого, так и для второго наборов коэффициентов затухания падает довольно быстро. К примеру, если учитывается только квадратичный коэффициент затухания, то на расстоянии 10 единиц при  $k_q = 1$  результирующая интенсивность на два порядка меньше исходной (т.е. она пренебрежимо мала). Таким образом, при выборе коэффициентов затухания следует соблюдать осторожность. Они оказывают сильное влияние на затухание. Я рекомендую ограничиться линейным фактором, выбрав его значение в интервале 0.001–0.0005.

### Результаты вычисления затухания для $k_c = k_l = k_s = 1$

Расстояние до источника d	$1/d$	$1/d^2$	$\frac{1}{1+1/d+1/d^2}$
0.500	2.000	4.000	0.571
1.000	1.000	1.000	0.333
1.500	0.667	0.444	0.211
2.000	0.500	0.250	0.143
2.500	0.400	0.160	0.103
3.000	0.333	0.111	0.077
3.500	0.286	0.082	0.060
4.000	0.250	0.063	0.048
4.500	0.222	0.049	0.039
5.000	0.200	0.040	0.032

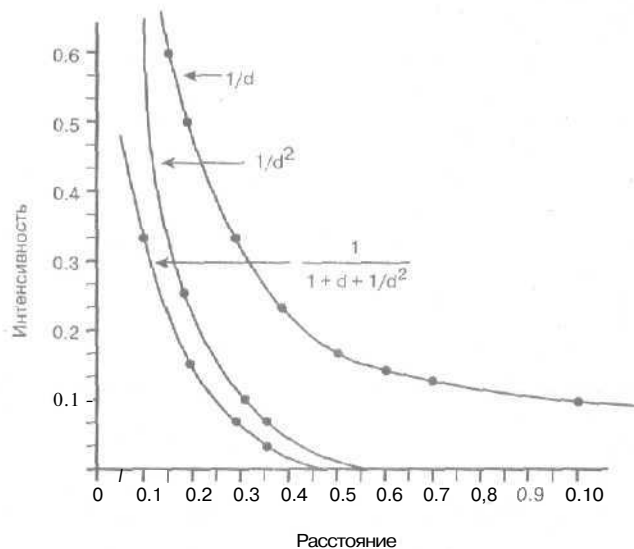


Рис. 8.12. Графики зависимости интенсивности освещения от расстояния до источника

Конечно же, когда все коэффициенты затухания отличны от нуля, интенсивность света убывает еще быстрее. Однако на практике оказывается, что упрощенная модель, в которой отсутствует квадратичный член, работает довольно хорошо;

$$I(d)_{\text{point}} = \frac{I_{0 \text{ point}} \cdot C_{\text{point}}}{k_c + k_l \cdot d}, \text{ где } d = |p - s|.$$

### Световые пятна

Последний вид источника света, который мы обсудим, — световое пятно (рис. 8.13). Моделирование световых пятен требует больших затрат вычислительных ресурсов (даже при использовании специального аппаратного обеспечения). Поэтому нам придется прибегнуть к

некоторым трюкам и упрощениям, чтобы хотя бы начать имитацию этих источников с помощью программных процессоров. Однако об этом позже, а пока что рассмотрим математические выражения, лежащие в основе подобных моделей. На рис. 8.14 представлена стандартная конфигурация с источником в виде светового пятна. Источник расположен в точке  $p$ , а луч от него, распространяясь в направлении  $I$ , попадает на поверхность  $S$  в точке  $s$ . Кроме того, имеется конус, в котором источник должен оказывать воздействие. Этот конус состоит из двух областей: внутренней и внешней. Внутреннюю область, которую мы обозначим через  $a$ , обычно называют *тенью* (umbra), а внешнюю (часть с мягким затенением), которая обозначается через  $\phi$ , — *полутенью* (penumbra).

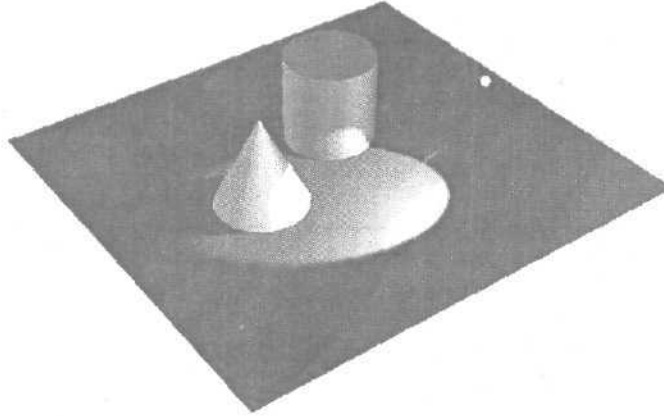


Рис. 8.13. Освещение объектов световым пятном

Интенсивность освещения во внутренней области разумно считать постоянной, а при переходе во внешнюю область она очень быстро уменьшается. Для моделирования освещения объектов, находящихся во внутренней и внешней областях светового конуса, источником которого является световое пятно, в большинстве процессоров используется описанный ниже алгоритм.

Случай 1. Если угол между вектором  $I$  и направлением на освещаемую точку  $s$  поверхности больше, чем угол внешнего конуса, данная точка не освещается.

Случай 2. Если угол между вектором  $I$  и направлением на освещаемую точку  $s$  поверхности находится во внутреннем конусе (тень), то используется 100% интенсивности источника; при этом учитывается затухание интенсивности с увеличением расстояния до источника.

Случай 3. Если угол между вектором  $I$  и направлением на освещаемую точку  $s$  поверхности находится во внешнем конусе (полутень), интенсивность света убывает не только с удалением от источника, но и при смещении точки к внешнему краю конуса.

Похоже, при моделировании световых пятен приходится выполнять много вычислений. Вопрос в том, нужны ли они на самом деле? Можно ли того же самого эффекта достичь с помощью упрощенной модели? Ответ — да, можно, и я скоро покажу, как это сделать. А пока что продумаем, как рассмотренную сложную модель можно было бы реализовать с помощью Direct3D, что станет для нас хорошей практикой.

Итак, чтобы смоделировать световое пятно, нужно задать его положение  $p$ , вектор направления на источник  $I$  и два угла ( $a$  и  $\phi$ ), определяющие величину внутреннего и внешнего конусов, соответственно. Кроме того, чтобы упростить процедуру сравнения величин углов, введем углы, величина которых равна половине исходных:  $\alpha^* = \alpha/2$  и  $\phi^* = \phi/2$ . И конечно же, нам понадобится цвет освещения  $C_{\text{spotlight}}$  и его интенсивность  $I_{\text{spotlight}}$ . Это просто ужас, сколько всего надо!

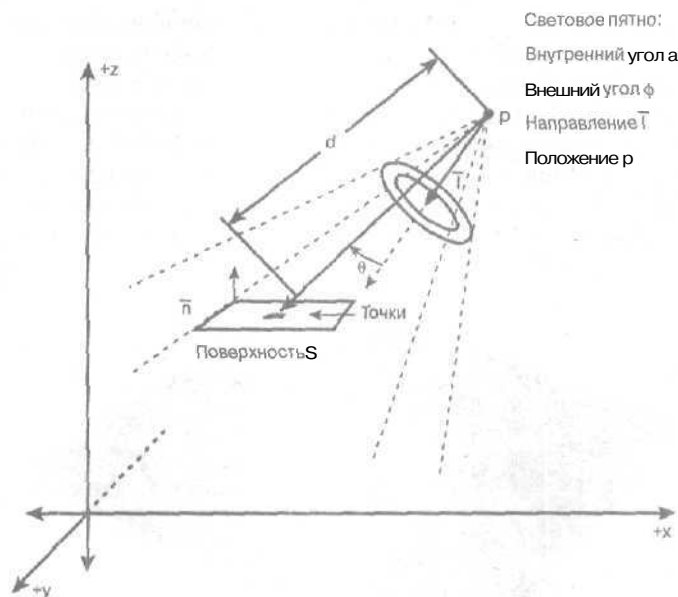


Рис. 8.14. Схема, лежащая в основе математической модели светового пятна

Получить представление о геометрических взаимоотношениях между элементами, из которых состоит модель светового пятна, поможет рис. 8.15. Суть в следующем: если угол  $\theta$  попадает за пределы светового конуса, соответствующая точка не освещается, если этот угол попадает во внутренний конус, то источник оказывает воздействие, причем его интенсивность должна быть постоянной во всем внутреннем конусе, когда же мы переходим во внешний конус, интенсивность света должна уменьшаться.

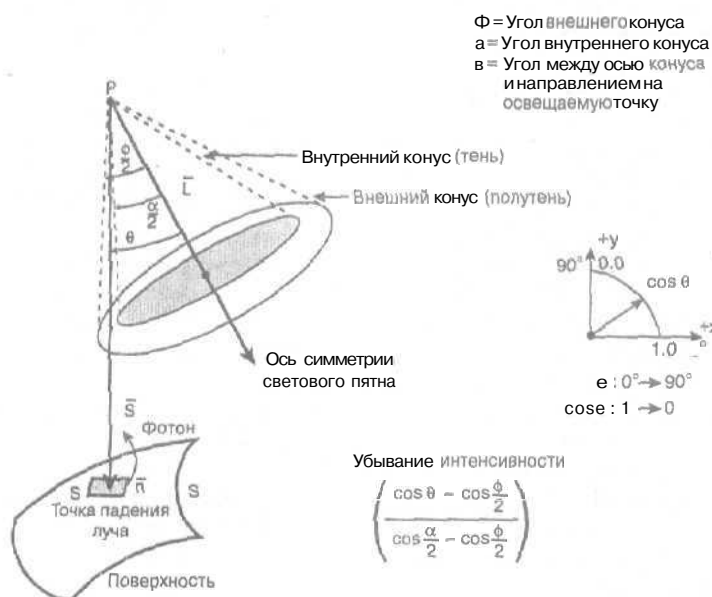


Рис. 8.15. Подробный анализ математической модели светового пятна

Наконец, понадобится показатель степени, определяющий степень сосредоточенности интенсивности источника, или коэффициент затухания, которым задается общая интенсивность светового пятна. Обозначим эту величину через  $pf$ . Наконец, поскольку световое пятно имеет определенное сходство с точечным источником, важно учесть убывание интенсивности освещения с увеличением расстояния от источника до освещаемого объекта, поэтому нам снова понадобятся коэффициенты затухания  $k_e$ ,  $k_l$  и  $k_q$ . С учетом сказанного, формула для интенсивности освещения, излучаемого световым пятном, выглядит следующим образом.

### Уравнение 8.15. Интенсивность светового пятна

Случай 1:  $\theta > \phi^*$  (за пределами светового конуса освещение отсутствует):

$$I(d)_{\text{spotlight}} = 0.$$

Случай 2:  $\theta < \alpha^*$  (во внутреннем конусе):

$$I(d)_{\text{spotlight}} = \frac{I0_{\text{spotlight}} \cdot Cl_{\text{spotlight}}}{k_e + k_l \cdot d + k_q \cdot d^2}, \text{ где } d = |p - s|.$$

Случай 3:  $\alpha^* < \theta < \phi^*$  (во внешнем конусе):

$$I(d)_{\text{spotlight}} = \frac{I0_{\text{spotlight}} \cdot Cl_{\text{spotlight}}}{k_e + k_l \cdot d + k_q \cdot d^2} \cdot \frac{(\cos \theta - \cos \phi^*)^{pf}}{(\cos \alpha^* - \cos \phi^*)}, \text{ где } d = |p - s|, \text{ а } pf = \text{показатель степени}$$

затухания.

Кто-то из вас может подумать, что все это несколько загадочно. Скалярное произведение — это еще куда ни шло, с ним можно мириться, но отношение косинусов выглядит подозрительно. Здесь я с вами полностью согласен. Рассмотрим эти математические выражения подробнее и убедимся, что они все же имеют смысл. В качестве примера рассмотрим конфигурацию, в которой угол  $\alpha$  равен  $40^\circ$ , угол  $\phi$  равен  $60^\circ$ , а угол  $\theta$  между внешним и внутренним конусами находится в интервале, который определяется двойным неравенством:  $20^\circ < \theta < 30^\circ$ . Подставив эти величины в правую часть формулы, по которой вычисляется интенсивность освещения, получим:

$$I = \frac{(\cos \theta - \cos 30^\circ)^{1.0}}{(\cos 20^\circ - \cos 30^\circ)}.$$

Чтобы упростить вычисления, я взял показатель степени, равный 1.0. А теперь начнем рассуждать: если величина угла стремится к нулю, то косинус этого угла стремится к единице; если же величина угла стремится к  $90^\circ$ , то косинус этого угла стремится к нулю. Нам нужно, чтобы приведенное выше отношение стремилось к единице по мере того, как величина угла  $\theta$  приближается к величине внутреннего угла (т.е. угла  $\alpha$ , который в данном случае равен  $40^\circ$ ; при этом мы приближаемся к области постоянной яркости). Также необходимо, чтобы это отношение стремилось к нулю по мере того, как величина угла  $\theta$  приближается к величине внешнего угла. Рассмотрим приведенное выше отношение разностей косинусов и подумаем, что с ним происходит при изменении угла  $\theta$ . Если  $\theta = 20^\circ$ , получаем:

$$I = \frac{(\cos 20^\circ - \cos 30^\circ)^{1.0}}{(\cos 20^\circ - \cos 30^\circ)} \sim 1.0.$$

Замечательно: это именно то, что нужно! А теперь предположим, что угол  $\theta$  равен внешнему углу конуса. При этом рассматриваемое выражение должно обратиться в ноль:

$$I = \frac{(\cos 30^\circ - \cos 30^\circ)^{1.0}}{(\cos 20^\circ - \cos 30^\circ)} \sim 0.0.$$

И снова получаем правильный результат. Возникает вопрос: как же выглядит кривая, описываемая этой формулой, в промежуточной области? Рассмотрим рис. 8.16, на котором изображены кривые для  $pf = 0.2$ ,  $1.0$  и  $5.0$ . Величины внутреннего и внешнего углов конуса выбраны равными  $30^\circ$  и  $40^\circ$  соответственно, а угол  $\theta$  меняется в интервале от  $15^\circ$  до  $20^\circ$  (поскольку это половинный угол, который отсчитывается от вектора  $L$ ). В большинстве случаев хорошо работает формула, описывающая спад интенсивности освещения, в которой в качестве показателя степени выбрана единица. В табл. 8.1 приведены данные, на основе которых построены графики на рис. 8.16.

**Таблица S. 1. Спад интенсивности освещения в зависимости от угла  $\theta$**

Показатель степени	Угол $\theta$	Спад
0.200000	15	1.000000
	16	0.961604
	17	0.912672
	18	0.845931
	19	0.740122
	20	0
1.000000	15	1.000000
	16	0.822205
	17	0.633248
	18	0.433187
	19	0.222083
	20	0.000000
5.000000	15	1.000000
	16	0.375752
	17	0.101829
	18	0.015254
	19	0.000540
	20	0.000000

Возникает вопрос: нужно ли все это? Все зависит от поставленных целей. Если мы занимаемся визуализацией не игровых трехмерных сцен — да, безусловно. Если же мы производим визуализацию трехмерных игр в реальном времени, возможно, изложенный выше материал не понадобится. В игре, события которой происходят в реальном времени, достаточно хорошие результаты дает любая, даже самая приблизительная модель светового пятна.

Теперь у нас есть все, что нужно для реализации простенькой модели светового пятна. Этим и займемся. Не станем разделять конус светового пятна на две области — внутреннюю и внешнюю. Вместо этого предположим, что спад интенсивности освещения происходит при любом отклонении от оси светового конуса. Кроме того, предположим, что этот спад пропорционален степени косинуса угла между осью конуса и тем направлением, в котором находится освещаемая точка. В данном случае степень, в которую возво-

дится косинус угла, отвечает за то, насколько сильно интенсивность освещения сосредоточена вокруг оси конуса. Такая модель позволит уменьшить количество условных операторов, а аналитическое выражение для нее имеет следующий вид:

$$I(d)_{\text{spotlight}} = \frac{I0_{\text{spotlight}} \cdot (1_{\text{spotlight}} \cdot \max((1 \cdot s), 0))^{pf}}{k_e + k_l \cdot d + k_s \cdot d^2}, \text{ где } d = |p - s|, \text{ а pf — показатель степени.}$$

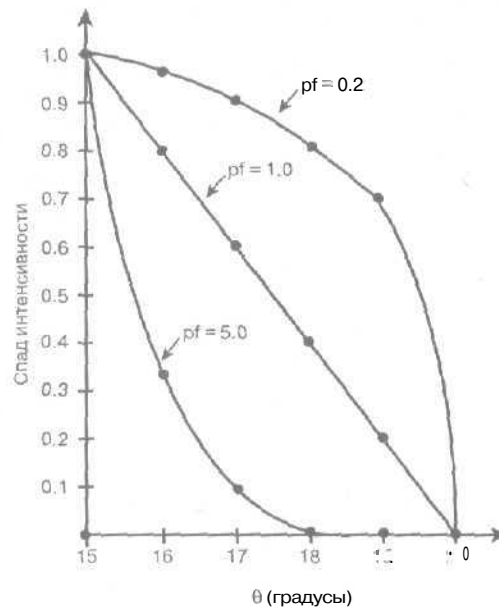


Рис. 8.16. Кривые отклика коэффициента, модулирующего зависимость интенсивности освещения от угла

## Резюме по моделированию освещения

Как видим, моделирование освещения требует больших усилий. При этом следует помнить о том, что моделирование освещения почти полностью сводится к определению цвета каждого из пикселей, из которых состоит многоугольник. До этого момента мы создавали модели виртуальных материалов, описывающие такие свойства этих материалов, как коэффициенты отражения для общего, диффузного и зеркального освещения. Кроме того, были рассмотрены модели виртуальных источников света, которые нужны для того, чтобы генерировать освещение виртуальных материалов. Эти источники могут быть простыми, например, общее освещение, которое распространяется повсюду, или сложными, например, световые пятна.

Пока что наш игровой процессор не поддерживает никаких материалов или источников освещения; над этим еще придется потрудиться. Со временем мы добавим эту поддержку, однако все по порядку. Теперь, когда вы получили представление о материалах и источниках освещения, я хочу, чтобы вы подумали обо всем этом, а тем временем пока заполним каркасы наших многоугольников — ведь нам все равно придется это сделать при добавлении надлежащего освещения.

### НА ЗАМЕТКУ

Все программы, структуры и другой код, встречающийся в этой главе, можно найти в файле `T3DLIB6.CPP`.

## Освещение и растеризация треугольников

В главе 8, "Растеризация векторов и двумерные преобразования", книги *Программирование игр для Windows, Советы профессионала* было описано, как выполнять растеризацию треугольников и многоугольников с помощью чистых цветов. В этой главе я собираюсь повторно изложить этот материал, но более основательно, чтобы подготовить вас к восприятию материала главы 9, "Интерполяционные методы затенения и аффинное отображение текстур". Для тех, кто не читал мою предыдущую книгу, я хочу кратко определить проблему. Во-первых, мы уже приняли соглашение, что разрабатываемый процессор трехмерной визуализации работает с треугольниками. Таким образом, нам не нужно беспокоиться о четырехугольниках или других более сложных многоугольниках. Рис. 8.17 приведен треугольник общего вида, который мы хотим растеризовать.



Рис. 8.17. Схема растеризации треугольника

Вывод треугольника очень напоминает вывод линии: нужно определить, какие пиксели образуют правую и левую стороны треугольника, а затем заполнить его строка за строкой. Этот процесс проиллюстрирован на рис. 8.17. Как видим, после вычисления наклона каждой из сторон можно переходить от одной строки развертки к другой, сдвигая при этом по оси  $x$  граничные точки ( $x_s$  и  $x_e$ ). Величина сдвига вычисляется, исходя из величины наклона. После этого проводится соединительная линия.

Алгоритм Брезенхама (Bresenham) нам не понадобится, поскольку мы не проводим линию. Нам нужно только узнать, в каких местах линия пересекает пиксель для каждой строки развертки. Ниже приводится алгоритм для случая, когда нижняя сторона треугольника параллельна строкам развертки.

1. Сначала вычисляем отношение  $dx/dy$  для левой и правой сторон треугольника. По сути, это величина, обратная наклону. С ней работать удобнее, поскольку мы собираемся использовать подход, в котором на основе вертикального сдвига нужно вычислить горизонтальный сдвиг. Таким образом, нам нужно знать изменение координаты  $x$  при каждом значении  $y$ , а это просто  $dx/dy$ . Назовем эти величины для левой и правой сторон треугольника  $dxy_{left}$  и  $dxy_{right}$ , соответственно.
2. Двигаясь от верхней вершины с координатами  $(x_0, y_0)$ , положим  $x_s = x_e = x_0$  и  $y = y_0$ .
3. Прибавим к  $x_s$  величину  $dxy_{left}$ , а к  $x_e$  — величину  $dxy_{right}$ . При этом получим границы интервала, который нужно заполнять.
4. Выведем линию, соединяющую точки  $(x_s, y)$  и  $(x_e, y)$ .
5. Вернемся к шагу 3 и будем выполнять итерации до тех пор, пока не дойдем до низа треугольника.

Конечно же, придется немного подумать над тем, чтобы правильно учесть начальные и граничные условия изложенного выше алгоритма, однако в целом он довольно прост.

Реализуем алгоритм растеризации треугольника, в котором нижняя сторона параллельна линии развертки, на основе чисел с плавающей запятой. Названия переменных, участвующих в алгоритме, примем такими, как приведено на рис. 8.17. Код алгоритма приведен ниже.

```
// Вычисление величин наклона.
float dxy_left = (x2-x0)/(y2-y0);
float dxy_right = (x1-x0)/(y1-y0);

// Присвоение начальных значений переменным, отслеживающим
// положение краев треугольника
float xs = x0;
float xe = x0;

// Вывод изображения для каждой строки развертки
for (int y=y0; y <= y1; y++)
{
    // Для текущей координаты y проводим цветом с прямую,
    // соединяющую точки xs и xe
    Draw_Line((int)xs, (int)xe, y, c);
    // Сдвиг на одну строку развертки вниз
    xs+=dxy_left;
    xe+=dxy_right;
} // for y
```

А теперь обсудим детали и отсутствующие части рассматриваемого алгоритма. При вычислении абсцисс конечных точек, ограничивающих треугольник в каждой линии развертки, в величинах этих абсцисс отбрасывается дробная часть. Возможно, это не самый лучший способ, поскольку при этом теряется определенная информация. Лучше было бы округлять координаты каждой конечной точки, добавляя к ним 0.5 перед тем, как привести их к типу int. Другая проблема возникает в связи с начальными условиями. Во время первой итерации в алгоритме выводится линия, длина которой равна одному пикселю. Это работает, однако оптимизация здесь не помешала бы.

Посмотрим, можно ли реализовать аналогичный алгоритм для треугольника, в котором верхняя сторона параллельна линии развертки. Для этого достаточно лишь по-другому обозначить вершины. Итак, теперь **p0** и **p1** — верхние вершины, а **p2** — нижняя. Далее нужно слегка изменить начальные условия алгоритма. После внесения изменений код выглядит следующим образом.

```
// Вычисление величин наклона.
float dxy_left = (x2-x0)/(y2-y0);
float dxy_right = (x2-x1)/(y2-y1);

// Присвоение начальных значений переменным, отслеживающим
// положение краев треугольника
float xs = x0;
float xe = x1;

// Вывод изображения для каждой строки развертки
for (int y=y0; y <= y2; y++)
{
    // Для текущей координаты y проводим цветом с прямую,
```

```

// соединяющую точки xs и xe
Draw_Line((int)(xs+0.5), (int)(xe+0.5), y, c);
// Сдвиг на одну строку развертки вниз
xs+=dxy_left;
xe+=dxy_right;
} // for y

```

Произвольный треугольник можно разбить на два одним из четырех способов.

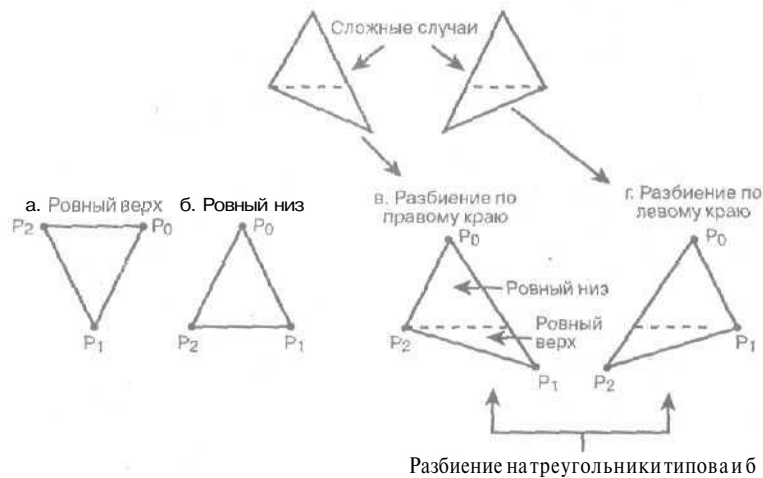


Рис. 8.18, Различные виды треугольников, подлежащих растеризации

Итак, у нас есть процедура растеризации треугольника, позволяющая выводить треугольники, у которых либо верхняя, либо нижняя сторона параллельна строке развертки. Теперь можно создать более общую процедуру, в которой произвольный треугольник представляется в виде двух треугольников, растеризацию которых мы уже умеем выполнять. После этого вызывается одна из представленных выше процедур. Процесс разложения произвольного треугольника на два треугольника специального вида проиллюстрирован на рис. 8.18. Эта простая функция, предназначенная для растеризации треугольника, уже находится в нашей библиотеке в различных формах, а ее различные прототипы показаны в представленном ниже коде.

```

// 8-битовая версия функции растеризации
void Draw_Triangle_2D(int x1,int y1, // Вершина 1;
                     int x2,int y2, // вершина 2;
                     int x3,int y3, // вершина 3;
                     int color,     // индекс цвета
                     // многоугольника;
                     UCHAR *dest_buffer, // целевой буфер;
                     int mempitch);     // шаг буфера

// 8-битовая версия функции растеризации, основанная на
// использовании чисел с фиксированной запятой
void Draw_TriangleFP_2D(int x1,int y1, // Вершина 1;
                       int x2,int y2, // вершина 2;
                       int x3,int y3, // вершина 3;

```

```

        int color,           // индекс цвета
                                // многоугольника;
        UCHAR *dest_buffer,  //целевой буфер;
        int mipmapitch);    // шаг буфера

// 16-битовая версия функции растеризации
void Draw_Triangle_2D16(int x1,int y1, // Вершина 1;
        int x2,int y2,           // вершина 2;
        int x3,int y3,           // вершина 3;
        int color,               // RGB-цвет
                                //многоугольника;
        UCHAR *dest_buffer,      //целевой буфер;
        int mipmapitch);        // шаг буфера

```

Приведенный выше код можно найти в файле T3DLIB1.CPP\H. Весь новый код, который будет появляться в данной главе, будет добавлен в файл T3DLIB6.CPP\H. Можно было бы, например, добавить 16-битовую версию функции растеризации для чисел с фиксированной запятой, однако я сомневаюсь, что это понадобится, — ведь нам пока что не нужно преодолевать никаких скоростных барьеров. К тому же нет никакой уверенности, что на современных компьютерах версия с фиксированной запятой работает быстрее, чем версия с плавающей запятой (хотя нет уверенности и в обратном).

Теперь, когда мы располагаем инструментом, позволяющим выводить на экран заполненные треугольники, добавим эту функциональную возможность в трехмерный процессор. Все, что для этого нужно, — взять функции, которые выводят каркасные объекты и списки визуализации, и изменить в них код, добавив вызовы функций, предназначенных для вывода треугольников. Так и сделаем. Ниже представлен полный список всех новых функций, которые были написаны для вывода заполненных объектов и списков визуализации (их параметры должны быть вам знакомы).

```

// Вывод 8-битового заполненного объекта
void Draw_OBJECT4DV1_Solid(OBJECT4DV1_PTR obj,
        UCHAR *video_buffer, int lpitch);

// Вывод 16-битового заполненного объекта
void Draw_OBJECT4DV1_Solid16(OBJECT4DV1_PTR obj,
        UCHAR *video_buffer, int lpitch);

// Вывод 8-битового списка визуализации
void Draw_RENDERLIST4DV1_Solid(RENDERLIST4DV1_PTR rend_list,
        UCHAR *video_buffer, int lpitch);

// Вывод 16-битового списка визуализации
void Draw_RENDERLIST4DV1_Solid16(RENDERLIST4DV1_PTR
        rend_list,
        UCHAR *video_buffer,
        int lpitch);

```

Начнем с функции, которая выводит объект OBJECT4DV1 в 8-битовом режиме. Ниже приведен код, выполняющий эту задачу.

```

void Draw_OBJECT4DV1_Solid(OBJECT4DV1_PTR obj,
        UCHAR *video_buffer, int lpitch)
{
    // Данная функция выводит на экран заполненный объект в
    // 8-битовом режиме. В ней не происходит удаление скрытых

```

```

// поверхностей и т.п. Эта функция лишь облегчает вывод
// объектов без их преобразования в набор многоугольников.
// В функции предполагается, что все координаты заданы в
// системе отсчета экрана. Однако в ней выполняется
// двумерное отсечение, причем оно выполняется поочередно
// для всех имеющихся в списке многоугольников, после чего
// каждый многоугольник выводится на экран
for (int poly=0; poly < obj->num_polys; poly++)
[
    // Данный многоугольник выводится тогда и только тогда,
    // если он не отсекается, не отбраковывается, если он
    // активен и видим. Однако заметим, что концепция
    // "обратной" поверхности для процессоров, работающих
    // с каркасными объектами, неприменима
    if (!(obj->plist[poly].state & POLY4DV1_STATE_ACTIVE) ||
        (obj->plist[poly].state & POLY4DV1_STATE_CLIPPED) ||
        (obj->plist[poly].state & POLY4DV1_STATE_BACKFACE))
        continue; // Переход к следующему многоугольнику.

    // Извлечение индексов вершин в основной список;
    // заметим, что многоугольники основаны на списке вершин,
    // который хранится в самом объекте
    int vindex_0 = obj->plist[poly].vert[0];
    int vindex_1 = obj->plist[poly].vert[1];
    int vindex_2 = obj->plist[poly].vert[2];

    // Вывод треугольника.
    Draw_Triangle_2D(obj->vlist_trans[ vindex_0 ].x,
        obj->vlist_trans[ vindex_0 ].y,
        obj->vlist_trans[ vindex_1 ].x,
        obj->vlist_trans[ vindex_1 ].y,
        obj->vlist_trans[ vindex_2 ].x,
        obj->vlist_trans[ vindex_2 ].y,
        obj->plist[poly].color,
        video_buffer, lpitch);
} // endfor poly
} // end Draw_OBJECT4DV1_Solid

```

Как видим, функция получилась довольно короткой. Все, что нужно было сделать, — извлечь список вершин многоугольника и передать в функцию вывода треугольника экранные координаты трех вершин, составляющих отображаемый треугольник.

#### СОВЕТ

Может возникнуть вопрос: а как же быть с отсечением? На данном этапе отсечение все еще выполняется в экранном пространстве, причем оно отлично от отбраковки трехмерных объектов. Следовательно, когда мы передаем многоугольники далее по цепочке функций, осуществляющих вывод на экран, функция растеризации экрана производит отсечение пикселя за пикселем, а это пока что нас вполне устраивает.

Рассмотрим версию функции, которая выводит список подлежащих визуализации завершенных объектов в 16-битовом формате.

```

void Draw_RENDERLIST4DV1_Solid16(RENDERLIST4DV1_PTR
    rend_list,
    UCHAR *video_buffer,
    int lpitch)

```

```

// Данная функция обрабатывает список подлежащих
// выводу объектов. Другими словами, она выводит
// все поверхности, которые находятся в списке
// каркасного объекта, в 16-битовом режиме. Заметим,
// что у нас нет необходимости сортировать
// многоугольники, однако в дальнейшем это
// понадобится, чтобы скрытые поверхности так и
// остались скрытыми. Задача определения битовой
// глубины и вызова подходящего растеризатора будет
// возложена на отдельную функцию

// Все что у нас есть на данном этапе - это список
// многоугольников; пришло время вывести их на экран
for (int poly=0; poly < rend_list->num_polys; poly++)
{
    // Данный многоугольник выводится тогда и только
    // тогда, когда он не отсекается, не отбраковывается,
    // если он активен и видим. Однако заметим, что
    // концепция "обратной" поверхности для процессоров,
    // работающих с каркасными объектами, неприменим
    if (!(rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_ACTIVE) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_CLIPPED) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV1_STATE_BACKFACE) )
        continue; // Переход к следующему многоугольнику

    // Вывод треугольника.
    Draw_Triangle_2D16(
        rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);
    } // for poly
} // Draw_RENDERLIST4DV1_Solid16

```

Как видим, приведенные выше функции занимают буквально несколько строк. Следующий наш шаг будет состоять в том, чтобы объединить модель **освещения** и вывод на экран. При этом изображение **многоугольников** будет формироваться исходя из того, из какого материала они сделаны и какое освещение на них падает.

## Подготовка к моделированию освещения

Теперь, **когда** мы располагаем математическими моделями материалов и источников освещения, рассмотрим, каким должен быть новый конвейер трехмерной **визуализации**, и попытаемся понять, в каком месте этого конвейера уместно ввести новый этап обработки (по крайней мере, абстрактно, поскольку в данной главе реализуется **достаточно**

простая задача). Как видно из рис. 8.19, новое звено в обновленном конвейере — это освещение. Данный этап будет выполняться после удаления из конвейера объектов, выходящих за поле зрения, а также исключения поверхностей, повернутых к зрителю обратной стороной. Таким образом, освещение будет моделироваться в “мировом” пространстве. Заметим, что с таким же успехом его можно реализовать в пространстве камеры. Единственное место, где освещение моделировать уже поздно, — после проецирования изображения на экран, поскольку в результате этого проецирования теряется информация о пространственном расположении объектов и трехмерная вселенная втискивается в плоскость.



Рис. 8.19. Новый конвейер трехмерной визуализации с добавленным этапом освещения

Я обращаю ваше внимание на то, какое положение в конвейере занимает этап освещения, поскольку мы собираемся заняться более конкретными вещами, а для этого нужно понять, что и когда нужно делать. Например, для моделирования освещения важно, каким образом направлены векторы нормали к многочисленным многоугольникам, из которых состоит освещаемая поверхность. Таким образом, перед выполнением вычислений, связанных с освещением, надлежит получить данные о направлении этих нормалей. Впрочем, освещение можно моделировать и после удаления поверхностей, направленных к наблюдателю обратной стороной, или на более поздних этапах, предварительно сохранив данные о направлении нормалей к участкам освещаемой поверхности.

Как видим, тут есть над чем подумать. Основной момент заключается в том, что мы хотим написать достаточно оптимизированный код, однако не потерять при этом общности и структурности программы. Если бы мы не стремились к этому, можно было бы просто программировать на ассемблере!

Прежде чем приступить к моделированию освещения многоугольников, из которых состоит система, поговорим о самих материалах и источниках освещения. С этим связано множество проблем, и первая из них — с тем форматом, который мы используем в данный момент. Нельзя сказать, что в формате PLG/PLX поддерживаются многие материалы (фактически в нем не поддерживается ни один материал). В этом формате имеются флаги для различных видов затенения (равномерного и плоского затенения, а также затенения по Гуро (Gouraud) и Фонгу (Phong)), однако никаких характеристик, которые хотя бы отдаленно напоминали все эти коэффициенты отражения материалов и другие подобные их характеристики. Кроме того, в нашей системе еще нет никаких источников освещения, поэтому для них также нужно организовать поддержку. Используемые нами на практике модели источников освещения и материалов будут предельно упрощенными. Это согласуется с моей философией, согласно которой нужно стремиться к простоте. В противном случае пришлось бы ждать, пока не появятся процессоры Pentium XII, чтобы программа заработала! Исходя из этого, приведем первый черновой вариант модели материала, которым будут покрыты поверхности многоугольников. Несмотря на то, что в дальнейшем нам может понадобиться лишь небольшая часть этой модели, она обладает определенной завершенностью.

## Моделирование материалов

Как уже упоминалось, материалы обладают множеством свойств, таких как коэффициенты отражения и цвет. Кроме того, нам понадобятся другие абстрактные характеристики материалов, с помощью которых можно будет смоделировать тип затенения, виды текстуры и другие подобные свойства. В текущей версии 1.0 имеется поле атрибута многоугольника, в которое заносится информация о нескольких характеристиках.

```
// Атрибуты многоугольников и их поверхностей
#define POLY4DV1_ATTR_2SIDED          0x0001
#define POLY4DV1_ATTR_TRANSPARENT    0x0002
#define POLY4DV1_ATTR_8BITCOLOR      0x0004
#define POLY4DV1_ATTR_RGB16          0x0008
#define POLY4DV1_ATTR_RGB24          0x0010

#define POLY4DV1_ATTR_SHADE_MODE_PURE 0x0020
#define POLY4DV1_ATTR_SHADE_MODE_CONSTANT 0x0020
// (псевдоним)
#define POLY4DV1_ATTR_SHADE_MODE_FLAT 0x0040
#define POLY4DV1_ATTR_SHADE_MODE_GOURAUD 0x0080
#define POLY4DV1_ATTR_SHADE_MODE_PHONG 0x0100
#define POLY4DV1_ATTR_SHADE_MODE_FASTPHONG 0x0100
// (псевдоним)
#define POLY4DV1_ATTR_SHADE_MODE_TEXTURE 0x0200

// Состояния многоугольников и их поверхностей
#define POLY4DV1_STATE_ACTIVE          0x0001
#define POLY4DV1_STATE_CLIPPED        0x0002
#define POLY4DV1_STATE_BACKFACE       0x0004
```

Вспомнили? А теперь учтем все это в моделях материалов. Мы не будем хранить все эти данные в каждом многоугольнике, а вместо этого определим в системе несколько материалов, которые затем в процессе визуализации нужно будет применить к различным многоугольникам. На рис. 8.20 проиллюстрированы взаимоотношения между материалами и многоугольниками, присущие новому определению. В такой модели твердого тела один и тот же объект можно заполнить несколькими различными материалами. В процессе визуализации игровой процессор будет просматривать список материалов, основываясь на их идентификаторе или указателе, и получать таким образом информацию о том, из какого материала изготовлен тот или иной многоугольник. При этом данную информацию не придется заносить в каждый объект, в состав которого входит данный многоугольник. Здорово придумано, правда?

Ниже приведена первая версия структуры, к которой мы пришли в процессе моделирования материалов. Конечно же, в наших силах ее изменить; в данной главе эта структура будет использоваться в минимальной степени, однако над ней полезно поразмышлять уже сейчас. Итак, приведем определение материалов.

```
// Определения материалов, наиболее подробно
// описывающие атрибуты наших многоугольников
#define MATV1_ATTR_2SIDED          0x0001
#define MATV1_ATTR_TRANSPARENT    0x0002
#define MATV1_ATTR_8BITCOLOR      0x0004
#define MATV1_ATTR_RGB16          0x0008
#define MATV1_ATTR_RGB24          0x0010
```

```

#define MATV1_ATTR_SHADE_MODE_CONSTANT 0x0020
#define MATV1_ATTR_SHADE_MODE_EMISSIVE 0x0020 //псевдоним
#define MATV1_ATTR_SHADE_MODE_FLAT 0x0040
#define MATV1_ATTR_SHADE_MODE_GOURAUD 0x0080
#define MATV1_ATTR_SHADE_MODE_FASTPHONG 0x0100
#define MATV1_ATTR_SHADE_MODE_TEXTURE 0x0200

```

```

//Состояние материалов
#define MATV1_STATE_ACTIVE 0x0001

```

```

//Определение системы материалов
#define MAX_MATERIALS 256

```

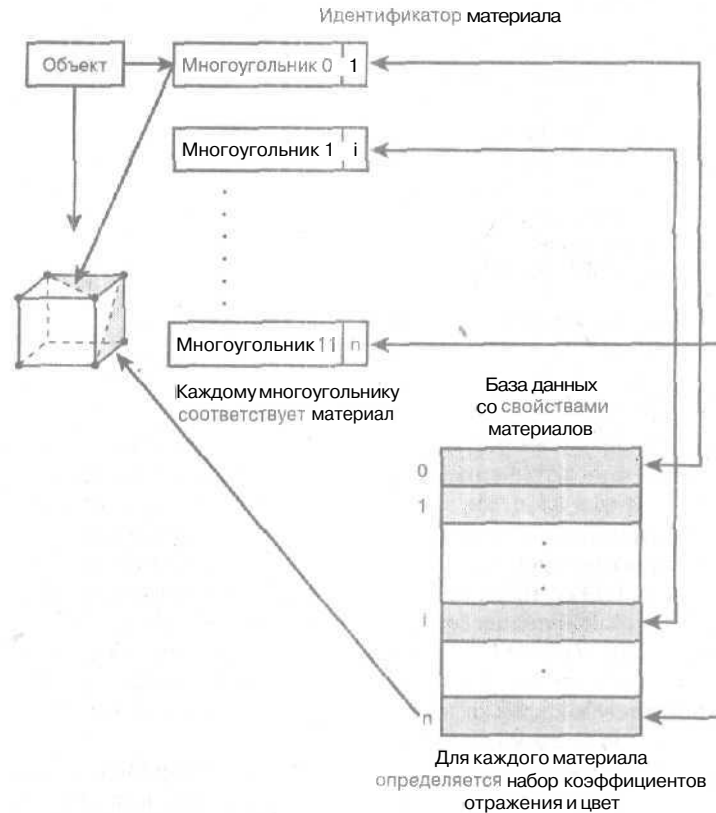


Рис. 8.20. Взаимоотношения между материалами и многоугольниками

А теперь приведем саму структуру, описывающую материалы. .

```

// Первая версия модели материала
typedef struct MATV1_TYP
{
    int state;           // Состояние материала
    int id;              // Идентификатор данного материала, -
                        // его индекс в таблице материалов
    char name [64];     // Название материала
}

```

```

int attr;    // Атрибуты, режимы затенения
             // (равномерное, плоское, по Гуро, по
             // Фонгу, окружение, текстуры и т.п.,
             // а также другие специальные флаги

RGBA_V1 color;    // Цвет материала;
float ka, kd, ks, power;    // коэффициенты общего,
                           // диффузного и зеркального отражения;
                           // обратите внимание, что они задаются
                           // в виде отдельных скаляров, поскольку
                           // этот формат используется во многих
                           // моделях вместе со степенью
                           // зеркального отражения

RGBA_V1 ra, rd, rs;    // Предварительно вычисленные
                       // произведения цвета на коэффициенты
                       // отражения, например, rd = color * kd;
                       // такие произведения в большей степени
                       // подходят для наших определений

char texture_file [80];    // Файл с текстурой
BITMAP texture;    // Текстурное отображение
                   // (если таковое имеется)
} MAT_V1, *MAT_V1_PTR;

```

В приведенном выше фрагменте кода нет ничего необычного. Поля `state`, `id`, `name` и `attr` я стараюсь вводить везде, где это возможно. А теперь обратите внимание на остальные поля, и в первую очередь на поле `color`. Фактически это объединение, в котором хранятся интенсивности красного, зеленого и синего цветов и значение альфа, определяющее прозрачность цвета, а также пара удобных методов, предоставляющих доступ к членам объединения.

```

// Цвет в формате RGB+alpha.
typedef struct RGBA_V1_TYPE
{
    union
    {
        int rgba;    // Сжатый формат;
        UCHAR rgba_M[4];    // формат массива;
        struct { UCHAR a,b,g,r; };    // формат с явным именем
    }; // union
} RGBA_V1, *RGBA_V1_PTR;

```

Кроме того, ниже приведен вспомогательный макрос, с помощью которого создается 32-битовое слово в формате `RGBA`.

```

// Создание 32-битового значения цвета в формате 8.8.8.8
// (8-битовый альфа-режим)
#define _RGBA32BIT(r,g,b,a)((a)<<24)+((b)<<16)+((g)<<8)+(r)

```

#### НА ЗАМЕТКУ

Ранее был определен макрос `_RGB32BIT(r,g,b,a)`, с помощью которого создаются 32-битовые слова, совместимые с технологией DirectX, но я не являюсь приверженцем альфа-формата. Тем не менее, бывают ситуации, в которых наличие значения альфа в младшем байте может оказаться полезным, поэтому далее будет использован формат `RGBA`.

Кроме того, в приведенной структуре содержатся значения коэффициентов отражения *ka*, *kd*, и *ks*, а также показатели степени. Дело в том, что многим разработчикам моделей нравится начинать обозначения коэффициентов отражения буквой *k*, поэтому мы тоже будем придерживаться этого соглашения. Однако при построении многих моделей освещения я использовал более удобные идентификаторы, начинающиеся с буквы *g*. Кроме того, в наших построениях мы, по сути, объединяем цвет и коэффициенты отражения, но в настоящей структуре данных они разделены. Тем не менее, в переменных *ga*, *rd* и *gs* хранятся копии коэффициентов отражения, умноженные на цвет. В дальнейшем эти величины могут нам понадобиться в том или ином виде, поэтому ничего страшного, если мы будем хранить их в обоих форматах. Ниже приведен пример использования скаляров *ka*, *kd* и *ks*, а также цвета в формате *ARGB* для вычисления величин *ga*, *rd* и *gs* (этот фрагмент кода взят из загрузчика объектов *Caligari trueSpace*, с которым мы ознакомимся далее в этой главе).

```
// Вычисление коэффициентов отражения материалов в формате,
// в котором произведено умножение на цвет
for (int rgb_index=0; rgb_index < 3; rgb_index++)
{
    // Коэффициент общего отражения
    materials[material_index+num_materials].
        ra.rgb_M[rgb_index] = ((UCHAR)
        (materials[material_index+ num_materials ].ka*
        (float)materials[material_index+ num_materials]
        .color.rgb_M [rgb_index] + 0.5));

    // Коэффициент диффузного отражения.
    materials[material_index+num_materials].
        rd.rgb_M[rgb_index] = ((UCHAR)
        (materials[material_index+ num_materials].kd *
        (float)materials[material_index+ num_materials]
        .color.rgb_M [rgb_index] + 0.5));

    // Коэффициент зеркального отражения.
    materials[material_index+num_materials].
        rs.rgb_M[rgb_index] = ((UCHAR)
        (materials[material_index+ num_materials].ks *
        (float)materials[material_index+ num_materials]
        .color.rgb_M [rgb_index] + 0.5));
} // for rgb_index
```

Наконец, в этом фрагменте кода поддерживается текстурное отображение и его имя.

А теперь, когда у нас есть структура, описывающая свойства материала, нам понадобится глобальная библиотека материалов, в которой эти структуры будут храниться. Это будет выглядеть следующим образом.

```
MATV1 materials[MAX_MATERIALS]; // Материалы, содержащиеся
                                // в системе
int num_materials;              // Номер текущего материала
```

Все очень просто: создаем новый материал, помещаем его в библиотеку, а затем — применяем к тому или иному многоугольнику! Конечно же, нужно еще создать многоугольники и игровой процессор для выполнения различных действий с материалами, однако этим мы займемся позже. А теперь создадим функцию, с помощью которой будет производиться инициализация библиотеки материалов, чтобы придать им заранее известное состояние.

```

int Reset_Materials_MATV1(void)
(
    // В этой функции для материалов устанавливается
    // исходное состояние.
    static int first_time = 1;

    // Если это первое применение, то все обнуляется
    if (first_time)
    {
        memset(materials, 0, MAX_MATERIALS * sizeof(MATV1));
        first_time = 0;
    } // if

    // Просмотр материалов и отключение всех текстур,
    // если таковые имеются
    for(int curr_matt = 0; curr_matt < MAX_MATERIALS;
        curr_matt++)
    {
        // Проверяем, связано ли с материалом свободное
        // текстурное отображение; проверка производится
        // независимо от того, активен ли материал
        Destroy_Bitmap(&materials[curr_matt].texture);
        // Теперь можно обнулять память.
        memset(&materials[curr_matt], 0, sizeof(MATV1));
    } // if

    return(1);
} // Reset_Materials_MATV1

```

Чтобы вернуть в исходное состояние массив материалов, входящий в состав библиотеки, достаточно просто вызвать функцию.

```
Reset_Materials_MATV1();
```

Далее рассмотрим пример того, как задать материал вручную, поскольку я не уверен, что есть смысл размышлять о вспомогательных функциях, пока мы не приступим к фактическому использованию материалов и не будем иметь в распоряжении формат, который их поддерживает. Предположим, нужно создать материал чисто синего цвета, который полностью отражает общий и диффузный свет, в котором используется только плоское затенение и нет никакой текстуры.

```

// Загрузка материала в слот 0
MATV1_PTR m = &Materials[0];

// Очистка структуры
memset(m, sizeof(MATV1));

m->state = MATV1_ACTIVE; // Состояние материала

m->id = 0; // Идентификатор материала

strcpy(m->name, "blue mat 1"); // Имя.
m->attr = MATV1_ATTR_2SIDED | MATV1_ATTR_16BIT |
    MATV1_ATTR_FLAT; // Атрибуты

```

```

m->color.rgb = _RGBA32BIT(0, Q, Z55, 255);
// Задаем синий цвет

m->ka = 1.0;
// Коэффициент общего отражения равен 1.0

m->kd = 1.0;
// Коэффициент диффузного отражения равен 1.0

m->ks = 1.0;
// Коэффициент зеркального отражения равен 1.0

m->power = 0.0; // Не используется, полагаем равным 0

// На самом деле это цвета в формате RGBA, поэтому нам
// понадобится умножить синий цвет на коэффициент отражения
// и сохранить его. Это можно сделать проще, но мы пойдем
// длинным путем в методических целях. Здесь приведен
// код только для синего канала
m->ra.b = (UCHAR)(m->ka * (float)m->color.b + 0.5);
m->rd.b = (UCHAR)(m->ka * (float)m->color.b + 0.5);
m->rs.rgb = 0; // Зеркальное отражение отсутствует

```

Конечно же, со временем мы займемся вспомогательными функциями, позволяющими делать все это автоматически, однако, поскольку мы пока что наверняка не знаем, как будут задаваться материалы (вручную или путем загрузки объектов), отложим разработку этих функций. А теперь пришло время заняться источниками освещения.

## Определение источников освещения

Практическая реализация источников освещения оказывается весьма утомительным и неприятным занятием. Пока дело не дошло до практического программирования, все выглядит в розовых тонах. Однако затем начинаешь понимать, что, к сожалению, мы попросту не располагаем какими-нибудь несколькими миллиардами тактов процессора для получения триллионов скалярных произведений и вычисления квадрильонов тригонометрических функций...

Возможно, я несколько преувеличиваю, однако надеюсь, что суть вы поняли. Вообще говоря, хороший игровой процессор, предназначенный для моделирования освещения, должен был бы иметь возможность имитировать бесконечное разнообразие источников света любого типа. Тогда соответствующие вычисления производились бы для каждого пикселя с учетом всех свойств материалов. Однако этого не будет. Даже самые совершенные ускорители трехмерной графики поддерживают сравнительно небольшое количество источников освещения (в большинстве случаев 8-16). Однако эту сложность можно обойти, например, путем отображения освещения, чем мы и займемся. Пока что следует иметь в виду, что мы не собираемся заниматься моделированием освещения на уровне отдельных пикселей. Будет создана поддержка только бесконечно удаленного и точечного источников. Более того, все вычисления, связанные с освещением, будут выполняться только для вершин треугольников, после чего будет производиться операция сглаживания освещения по поверхности (затенение по Гуро или по Фонгу).

Теперь, когда я разрушил ваши мечты о программном решении, выполняющем затенение для каждого пикселя отдельно, определим структуру, в которой задаются основные характеристики источника освещения. Эта структура способна будет поддерживать ис-

точники трех рассмотренных ранее видов, в том числе и световое пятно, хотя при дневном свете оно имеет весьма незначительные шансы быть заметным.

Ниже приведены программные определения источников **освещения** (версия 1.0). Обратите внимание, что реализованы световые пятна двух типов, чтобы в дальнейшем у нас была возможность пользоваться стандартной или упрощенной моделью.

```
// Определение видов источников освещения
#define LIGHTV1_ATTR_AMBIENT 0x0001 // Общее освещение
#define LIGHTV1_ATTR_INFINITE 0x0002 // Бесконечно
// удаленный источник
#define LIGHTV1_ATTR_POINT 0x0004 // Точечный источник
#define LIGHTV1_ATTR_SPOTLIGHT1 0x0008 // Световое пятно 1
// (упрощенное)
#define LIGHTV1_ATTR_SPOTLIGHT2 0x0010 // Световое пятно 2
// (сложное)

#define LIGHTV1_STATE_ON 1 // Включение света
#define LIGHTV1_STATE_OFF 0 // Выключение света

#define MAX_LIGHTS 8 // Хотя бы с одним
// получилось!
```

Нужно иметь возможность выбирать тип источника и задавать его состояние, т.е. включать его и выключать. Приведем структуру данных, в которой хранятся параметры источника.

```
// Структура с параметрами источника, версия 1.0
typedef struct LIGHTV1_TYP
{
    int state; // Состояние источника
    int id; // Идентификатор источника
    int attr; // Тип источника и дополнительные
    // атрибуты

    RGBAV1 c_ambient; // Интенсивность общего освещения
    RGBAV1 c_diffuse; // Интенсивность диффузного освещения
    RGBAV1 c_specular; // Интенсивность зеркального освещения
    POINT4D pos; // Положение источника
    VECTOR4D dir; // Направление источника
    float kc, kl, kq; // Степени затухания
    float spot_inner; // Внутренний угол светового пятна
    float spot_outer; // Внешний угол светового пятна

    float pf; // Показатель степени, определяющий
    // спад интенсивности светового пятна
} LIGHTV1, *LIGHTV1_PTR;
```

Следует помнить о том, что в процессе фактической визуализации можно использовать и материалы, и источники **освещения**. Конкретное применение зависит от того, какие действия производятся при помощи игрового процессора. На данной ранней стадии мы просто пытаемся создать модели, которые понадобятся в дальнейшем, поэтому нужно постараться не сделать **чего-нибудь** нелепого и ненужного. При создании трехмерных игровых процессоров нельзя недооценивать важность этапа планирования. В процессе разработки нужно продумать, как будут взаимодействовать другие части программы с игровым процессором. Если отложить это продумывание того момента, когда придется

приступать к разработке этих самых других частей, то цена такой недалёковидности будет слишком велика. Первый пример, который приходит в голову, — **добавление** возможности играть в игру по сети после того, как работа над процессором уже завершена!

Но хватит философии. Создадим источники **освещения**. Я **думаю**, что над источниками, в отличие от материалов, не стоит долго размышлять. Они либо есть, либо их нет, и игровой процессор может решать, в какой степени их использовать. Однако вспомогательную функцию для создания источников освещения, которая будет работать без большого количества модификаций, придется создавать наверняка. По **сути**, нам понадобится задать тип источника, его положение, направление и другие величины, в зависимости от того, источник какого типа создается. Кроме **того**, поскольку число источников будет небольшим (4-8), мы будем придерживаться установки, согласно которой можно будет включать любой из источников, входящих в массив. Затем игровой процессор будет проходить по всему массиву источников и обрабатывать те из них, которые включены. Здорово? Здорово! Начнем с того, что создадим вспомогательную функцию, позволяющую инициализировать все источники света и выключать их.

```
int Reset_Lights_LIGHTV1(void)
{
    // Эта функция устанавливает исходные значения для
    // всех источников освещения, входящих в систему
    static int first_time = 1;

    memset(lights, 0, MAX_LIGHTS*sizeof(LIGHTV1));
    // Обнуление количества источников
    num_lights = 0;

    // Обнуление счетчика
    first_time = 0;

    // Возврат кода успешного выполнения
    return(1);
} // Reset_Lights_LIGHTV1
```

Таким образом, в разделе инициализации, **входящем** в состав игры или процессора, достаточно вызвать функцию `Reset_Lights_LIGHTV1()`. Продолжим. Ниже приведена вспомогательная функция, позволяющая создавать источник.

```
int Init_Light_LIGHTV1(
    int      index,          // Индекс создаваемого источ-
                           // ника (0..MAX_LIGHTS-1)
    int      _state,         // Состояние источника
    int      _attr,          // Тип и дополнительные
                           // квалификаторы источника
    RGBA_VI _c_ambient,      // Интенсивность общего
                           // освещения
    RGBA_VI _c_diffuse,      // Интенсивность диффузного
                           // освещения
    RGBA_VI _c_specular,     // Интенсивность зеркального
                           // освещения
    POINT4D_PTR _pos,        // Положение источника
    VECTOR4D_PTR _dir,       // Направление источника
    float    kc, kl, kq,     // Коэффициенты затухания
    float    _spot_inner,    // Внутренний угол светового
```

```

float    _spot_outer,    // пятна
                                // Внешний угол светового
                                // пятна
float    _pf)            // Показатель степени, опре-
                                // деляющий спад интенсив-
                                // ности освещения
}

// 6 этой функции на основе флагов, переданных
// переменной _attr, инициализируется источник.
// Ненужные значения обнуляются

// Проверим, не выходит ли интенсивность за
// допустимые рамки
if (index < 0 || index >= MAX_LIGHTS)
    return(0);

// Все хорошо; инициализируем параметры источника
// (многие поля могут оказаться ненужными)
lights[index].state = _state; // Состояние источника
lights[index].id = index; // Идентификатор источника
lights[index].attr = _attr; // Тип и дополнительные
                                // атрибуты источника

lights[index].c_ambient = _c_ambient;
// Интенсивность общего освещения

lights[index].c_diffuse = _c_diffuse;
// Интенсивность диффузного освещения

lights[index].c_specular = _c_specular;
// Интенсивность зеркального освещения.
lights[index].kc = _kc;
// Постоянный, линейный и квадратичный
// коэффициенты затухания
lights[index].kl = _kt;
lights[index].kq = _kq;

if (pos)
    VECTOR4D_COPY(&lights[index].pos, _pos);
// Положение источника
if (dir)
{
    VECTOR4D_COPY(&lights[index].dir, _dir);
// Направление источника

    // Нормализуем
    VECTOR4D_Normalize(&lights[index].dir);
} // if

lights[index].spot_inner = _spot_inner;
// Внутренний угол светового пятна

lights[index].spot_outer = _spot_outer;
// Внешний угол светового пятна

```

```

lights[index].pf      = _pf;
// Показатель степени, определяющий спад
// интенсивности освещения

// Возврат индекса источника при успешном выполнении
return (index);

> // Create_Light_LIGHTV1

```

В этой функции переданные в нее параметры после некоторых проверок просто копируются в специальную структуру. После того как функция завершит работу, она возвращает индекс только что созданного источника света. Конечно, этот индекс тоже передается в функцию, но в этом нет ничего страшного. Создадим некоторые источники света с помощью имеющейся у нас функции...

### Создание источника общего света

Общий источник применяется чуть ли не в каждом эпизоде игры. Фактически, в нашем первом процессоре будет только общее освещение. Оно задается следующим образом.

```

ambient_light = Init_Light_LIGHTV1(
    0, // Для общего света
        // используется
        // индекс 0
    LIGHTV1_STATE_ON, // Включение источника
    LIGHTV1_ATTR_AMBIENT, // Тип источника
    _RGBA32BIT(255,255,255,0), // Чисто белый цвет (только
    0, 0, // для общего источника)
    NULL, NULL, // Положение и направление
        // не задается
    0,0,0, // Спада интенсивности нет
    0,0,0); // Информации, относящейся к
        // световому пятну, нет

```

Просто, не так ли? После того, как источник включен, можно вручную выключить его, осуществив доступ к переменной состояния.

```
Lights[ambient_light].state = LIGHTV1_STATE_OFF;
```

**СОВЕТ**

Можно было бы заниматься вспомогательными функциями все дни напролет, однако я стремлюсь идти простыми путями; в противном случае я бы программировал на Java :-)

### Создание бесконечно удаленного источника (направленный свет)

Теперь давайте создадим источник бесконечно удаленного диффузного освещения, подобного тому, каким является Солнце. Итак, зададим для него желтый цвет, а направление — как показано на рис. 8.21. Возможно, технически удобнее было бы работать с источником, направленным вниз, однако это просто вектор, и такое его направление упрощает математические преобразования.

```
VECTOR4D sun_dir = {0, -1, 0, 0};
```

```

sun_light = Init_Light_LIGHTV1(
    1, // Для солнечного света
        // используется индекс 1
    LIGHTV1_STATE_ON, // Включение света.
    LIGHTV1_ATTR_INFINITE, // Тип источника.

```

```

0, // Чисто желтый цвет
RGBA32BIT(255,255,0,0), // только для
0, // диффузного члена
NULL, &sun_dir, // Положение не задается, а
// направление - (0,-1,0).
0,0,0, // Затухание не нужно
0,0,0); // Информации, относящейся к
// световому пятну, нет

```

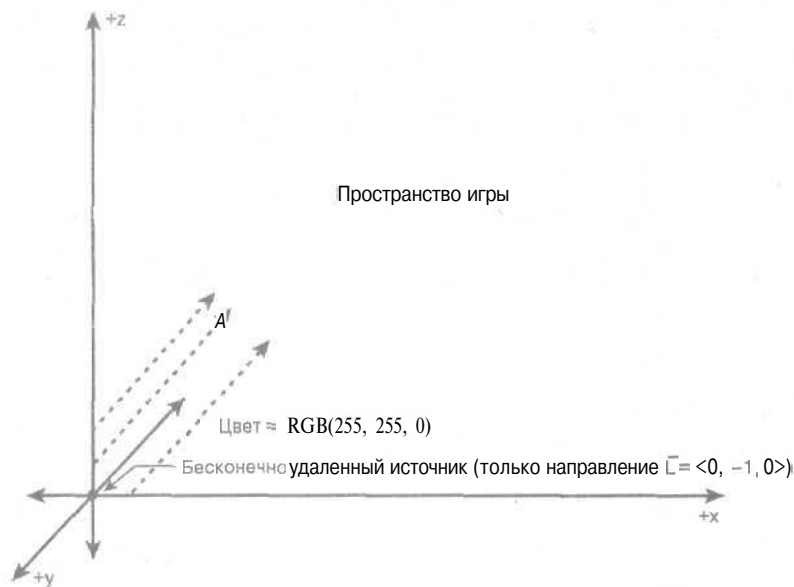


Рис. 8.21. Расположение бесконечно удаленного источника для рассматриваемого примера

## Создание точечного источника

Моделируя точечный источник, используем для него параметры модели солнечного света, причем расположим его на оси y на расстоянии, равном 10000 единиц (поскольку для точечного источника нужно задавать положение). Это проиллюстрировано на рис. 8.22.

```
VECTOR4D sun_pos = {0, 10000, 0, 0};
```

```

sun_light = Init_Light_LIGHTV1(1, // Для солнечного света
                                // используется индекс 1
                                LIGHTV1_STATE_ON, // Включение света.
                                LIGHTV1_ATTR_POINT, // Тип источника.
                                0, // Чисто желтый
                                RGBA32BIT(255,255,0,0), // цвет только
                                0, // для диффузного члена
                                &sun_pos, NULL, // Положение по оси y,
                                // направление не задается.
                                0,1,0, // Линейное затухание - 1
                                0,0,0); // Информации, относящейся к
                                // световому пятну, нет

```

## Создание светового пятна

Наконец, давайте создадим световое пятно просто для того, чтобы лучше запомнить, как вызывается данная функция! Если же говорить серьезно, то, как видно из модели световых пятен, их полное моделирование в реальном времени настолько сложное, что даже не обсуждается. Более того, мы имеем возможность выполнять вычисления интенсивности освещения для светового пятна только в вершинах многоугольника, поскольку делать это для каждого пикселя безрассудно. Таким образом, чтобы освещение от светового пятна можно было увидеть, понадобится множество многоугольников, входящих в состав объекта. Это не означает, что для светового пятна даже нельзя создать демонстрационный пример, просто использовать источник такого типа в игре будет весьма затруднительно — при этом работа программы слишком сильно замедляется. С другой стороны, существуют различные трюки, такие как отображение освещения. Мы к ним еще обратимся в этой книге, так что не падайте духом.

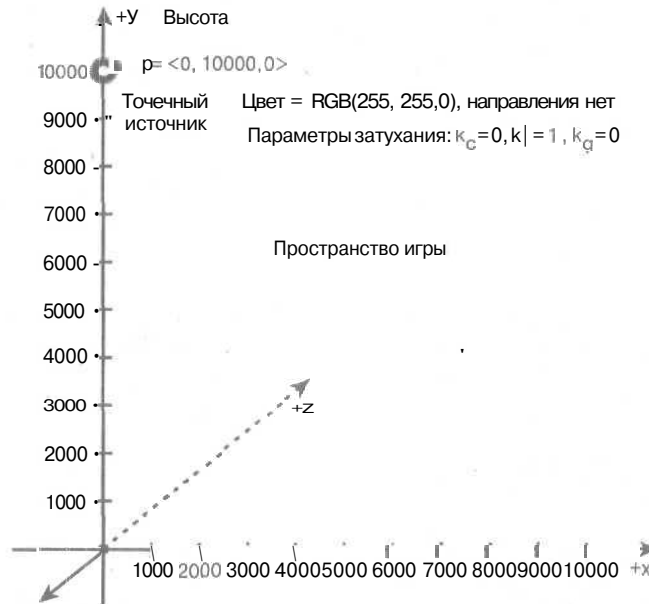


Рис. 8.22. Расположение точечного источника для данного примера

Давайте создадим световое пятно, которое расположено в точке с координатами  $p = (1000, 1000, -1000)$  и светит в направлении начала координат (следовательно, направление задается вектором  $v = \langle -1, -1, 1 \rangle$ ). Зададим для этого источника белый цвет с интенсивностью, равной 50% от максимальной:  $RGB = (128, 128, 128)$ . Пусть данный источник влияет только на материалы, у которых отличны от нуля коэффициенты диффузного и зеркального отражения. Определим внутренний угол светового конуса равным  $30^\circ$ , а внешний —  $60^\circ$ . Показатель степени, определяющий затухание источника, будет равен 1.0. Одним словом, забот полон рот! Ниже приведен код, необходимый для определения описанного выше светового пятна, а расположение самого пятна проиллюстрировано на рис. 8.23.

```
VECTOR4D spot_pos = {1000, 1000, -1000, 0};  
VECTOR4D spot_dir = {-1, -1, 1, 0};  
float umbra = 30, penumbra = 60, falloff = 1.0;
```

```

int spot_light = Init_Light_LIGHTV1(
    5, // Для данного источника
        // задаем индекс 5
    LIGHTV1_STATE_ON, // Включение источника
    LIGHTV1_SPOTLIGHT1, // Тип источника (пятно 1)
    0, // Белый цвет половинной
    _RGBA32BIT(128,128,128,0), // интенсивности только для
    0, // диффузного члена
    &spot_pos, &spot_dir, // Положение источника
    umbra, penumbra, falloff); // Информация, относящаяся
        // к пятну

```

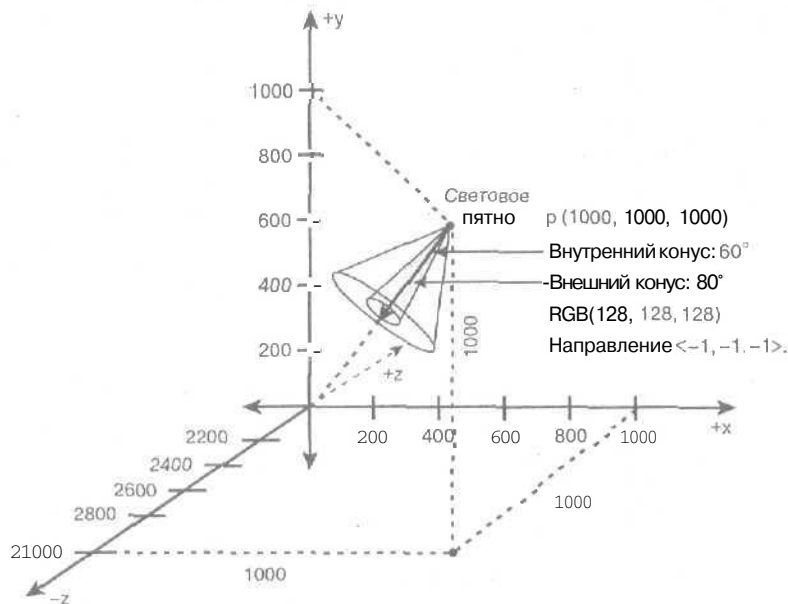


Рис. 8.23. Расположение светового пятна в данном примере

## Затенение в реальном мире

Мы почти готовы к тому, чтобы вывести что-нибудь на экран! Забавно, что половина всего, что мы узнали в этой главе, не пригодится в дальнейшем. Но, как уже было сказано, материалы и источники освещения — это такие вещи, о которых следует иметь представление до того, как мы приступим к их изучению... А теперь я хотел бы обсудить частные случаи реализации и оптимизации применительно к 8-битовому и 16-битовому графическим режимам. Это рассмотрение полезно по той причине, что 8-битовый режим предположительно работает в три раза быстрее, чем 16-битовый, поскольку в нем нет необходимости обрабатывать три канала — вычисления производятся только для одной величины, соответствующей яркости или светимости. Поговорим немного об этом.

### 16-битовое затенение

Затенение начинается с материалов и источников освещения. Рассмотренные до сих пор модели в основном базируются на формате RGB. Например, коэффициенты отражения мате-

риалов задаются в терминах RGB, цвета источников освещения задаются в формате RGB и т.д. Таким образом, в большинстве случаев связанные с **освещением** вычисления можно выполнять именно так, как это было описано в математической модели в начале данной главы. Конечно же, в конечной реализации допустимы трюки или дополнительные предположения.

Например, для ускорения работы программы можно было бы предположить, что все источники освещения излучают белый свет. Можно было бы также прийти к заключению, что на все объекты падает только общее и диффузное освещение. Однако что бы мы ни делали, рано или поздно наступает момент, когда нужно вычислять конечный цвет многоугольника, вершины или пикселя. Это означает, что придется выполнять определенные математические преобразования. Однако не следует забывать, что мы стремимся свести количество таких преобразований к минимуму. Здесь стоит заметить, что **выполнение** каких-либо действий на уровне многоугольников или вершин дает шансы получить работоспособную программу, а выполнение этих действий на уровне отдельных пикселей сводит эти шансы к нулю.

Здесь может возникнуть вопрос: в каком формате следует выполнять математические преобразования и почему? Мы будем использовать 16-битовый RGB-режим. Это означает, что в определенный момент придется произвести переход в этот режим. Нужно решить, выполнять ли вычисления с учетом того, что в дальнейшем предстоит указанное выше преобразование, или выполнять их в более общей форме, что позволит избежать его. Я хотел бы, чтобы вы поразмышляли над подобными **вопросами**. На них есть только один хороший ответ: используйте тот метод, который работает быстрее.

## 8-битовое затенение

С точки зрения математики, 16-битовое затенение проще, чем 8-битовое. Причина этого, конечно же, заключается в том, что в 16-битовом режиме выполняется понятное однозначное преобразование интенсивности в формате RGB для каждого пикселя экрана. Например, пусть у нас есть многоугольник чисто синего цвета ( $RGB(0,0,255)$ ), обладающий диффузным и общим отражением. Теперь предположим, что интенсивность освещения снижается на 50% для активного освещения и на 20% — для **общего**.

Имеется несколько подходов к решению подобных задач. Во-первых, поговорим об общем освещении. Если материал синего цвета, а освещение — красного, то мы ничего не увидим на экране, если будем работать только в пространстве RGB. Дело в том, что

```
Final_pixelrgb = materialrgb * ambientrgb =  
= (0,0,255) * (1,0,0,0) = (0,0,0)
```

Однако можно придерживаться упрощающего предположения, согласно которому все источники света излучают белый цвет различной интенсивности. В этом случае на экране все же будет какое-то изображение.

```
Final_pixelrgb = materialrgb * ambientrgb =  
= (0,0,255) * (1,0,1,0,1,0) = (0,0,25)
```

### НА ЗАМЕТКУ

Обратите внимание, что преобразование интенсивности света выполняется с помощью чисел с плавающей точкой в интервале (0..1). В итоге **результат не выходит за пределы** цветового пространства. Интенсивность можно было бы задавать в интервале (0..255), но потом пришлось бы выполнить **соответствующее** преобразование результата.

## Сложная RGB-модель для 8-битового режима

Следует иметь в виду, что в завершающей версии разрабатываемого процессора для облегчения **вычислений** в основном будет использоваться белое освещение. Теперь перед

нами возникает проблема. Во всех рассмотренных ранее примерах конечный цвет пикселя представлялся в 24-битовом RGB-формате или в формате (0..1, 0..1, 0..1), который легко преобразовать в 24-битовый RGB-формат. Дело в том, что далее мы преобразуем последний в 16-битовый формат, а затем выведем изображение на экран. Непонятно, как выполнять подобное преобразование в 8-битовом режиме. Ведь если система работает в этом режиме, то она не имеет ничего общего с форматом RGB; вместо этого в данном режиме все цвета проиндексированы, что и приводит к проблеме. Как же найти решение? Как моделировать освещение в режиме с проиндексированными цветами?

Ответ очень прост. Нужно ввести дополнительный шаг, который будет заключаться в преобразовании RGB-цветов в цветовые индексы, и только *после* этого выводить изображение на экран. Конечно же, при этом возникает еще одна проблема. Ведь в 8-битовом режиме может быть всего 256 цветов, поэтому в процессе преобразования нужно решать, какой цвет из палитры подходит в данном случае больше других, после чего использовать в результирующем изображении индекс именно этого цвета. Этот процесс проиллюстрирован на рис. 8.24. Пусть, например, у нас есть цвет RGB(10, 45, 34). Ближе всего к нему в палитре расположен цвет с индексом 45, которому соответствует RGB-значение (14, 48, 40) — именно его и следует возратить в качестве преобразованной величины. Однако это означает, что для каждого пикселя пришлось бы находить наименьшее квадратичное отклонение по всем цветам, из которых состоит палитра, а это большая работа! Таким образом, нужно создать таблицы соответствия, которые бы помогли выполнять преобразование из одного режима в другой. Для этого следует решить, какому цвету в формате RGB соответствует тот или иной индекс, и занести результаты в таблицу.

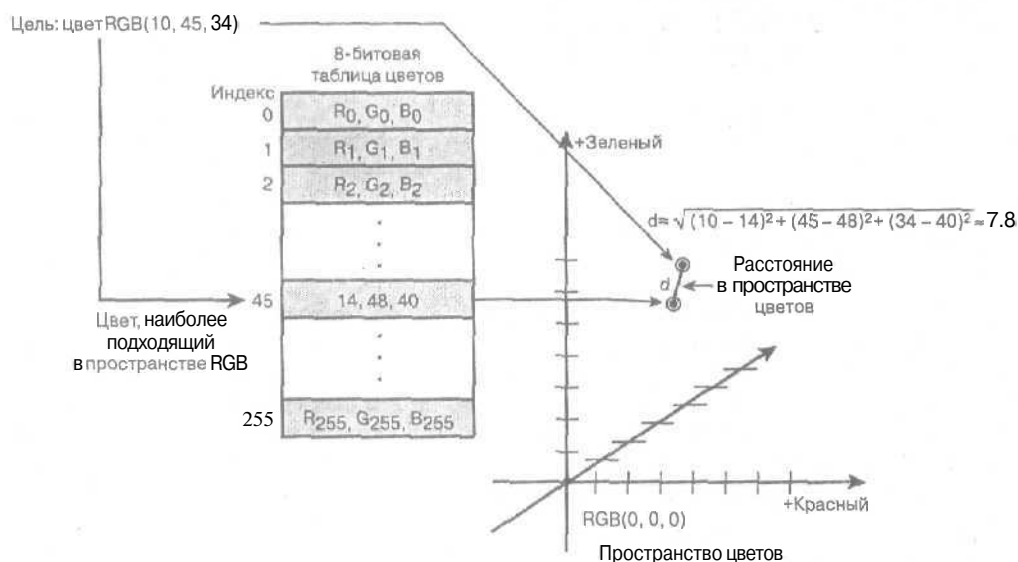


Рис. 8.24. Поиск наиболее подходящего цвета в палитре

#### СОВЕТ

В процессе применения 8-битового цветового режима с затенением в реальном времени необходимо использовать цветовую палитру, покрывающую большую часть RGB-пространства и обладающую множеством уровней яркости. При неудачном выборе палитры возникает риск получить изображение низкого качества. Рекомендую выбирать палитры, предоставленные в файлах PALDATAxx.BMP|PAL и хорошо себя зарекомендовавшие. Конечно, поскольку в 8-битовых режимах предусмотрена одна палитра, именно эти цвета, выбранные вами, и будут использоваться во всей игре.

В каком бы режиме **не** выполнялись вычисления — в режиме RGB 8.8.8 или в 16-битовом режиме (в формате 5.5.5 или 5.6.5) — мы все равно не сможем создать таблицу, отображающую в индексы все значения, доступные в этих режимах. Такая таблица заняла бы 4 Гбайт, что представляет собой немалый объем. Однако можно создать таблицу соответствия для преобразования цветов из 16-битового формата RGB в 8-битовый индекс. Такая таблица займет всего 64 Кбайт памяти — это вполне допустимо! Заметим, что поскольку исходное изображение может быть либо в формате 5.6.5, либо в формате 5.5.5, нужно иметь возможность генерировать одну из двух возможных таблиц (каждую для своего входного формата). Следует также иметь в виду, что если исходное изображение будет в 24-битовом формате RGB, то сначала нужно преобразовать его в 16-битовый формат 5.5.5 или 5.6.5. Затем это значение будет использовано как единое бинарное слово, представляющее собой индекс в состоящей из однобайтовых элементов таблице цветов, каждый индекс которой указывает на цвет 256-цветовой палитры, ближайший к данному RGB-значению. Этот процесс проиллюстрирован на рис. 8.25.

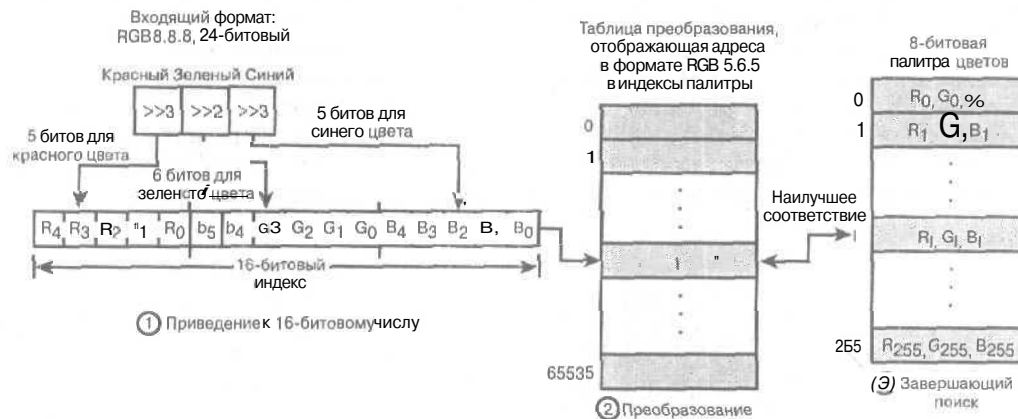


Рис. 8.25. Структура преобразования проиндексированных RGB-цветов с помощью таблицы соответствия

Из вышесказанного следует, что нам нужна функция, которая работает таким образом: на входе ей указывается 16-битовый формат, для которого необходимо сгенерировать таблицу преобразования цветов, исходную 8-битовую палитру и указатель на таблицу, которая будет заполняться данными. Возможно, нам потребуется несколько палитр и таблиц соответствия, отвечающих разным эпизодам игры. В противном случае данная функция будет использовать одну глобальную таблицу. Ниже приведен код функции, выполняющей описанные выше действия.

```
int RGB_16_8_IndexedRGB_Table_Builder(
    intrgb_format, // Формат, для которого строится таблица
    LPPALETTEENTRY src_palette, // Исходная палитра
    UCHAR* rgblookup) // Таблица соответствия
{
    // В данную функцию передается формат RGB, для которого
    // нужно сгенерировать таблицу соответствия;
    // dd_pixel_format=DD_PIXEL_FORMAT565 или
    // DD_PIXEL_FORMAT555
    // Обратите внимание, что в формате 5.5.5 входные
    // значения занимают 32 Кбайта памяти, и старший бит
```

```
// будет настроен так, чтобы для таблицы соответствия в
// любом случае потребовалось всего 32 Кбайта.
// Вызывающая программа должна выделить память для
// таблицы соответствия и передать ее в функцию. В самой
// функции память не выделяется. Для всех RGB-цветов в
// 16-битовом пространстве отображение в дискретной
// пространство RGB 8-битовой палитры производится по
// методу наименьших квадратов
```

```
// Сначала проверим указатели
if (!src_palette || !rgblookup)
    return(-1);
```

```
// Для какой глубины цвета генерируется таблица?
if (rgb_format==DD_PIXEL_FORMAT565)
```

```
{
    // Все элементы занимают 64 Кбайта памяти. Создадим
    // цикл для их поиска. Сведем объем работы к
    // минимуму, что важно даже для процессора Pentium.
    // Всего здесь 65536*256 итераций!
    for (int rgbindex = 0; rgbindex < 65536; rgbindex++)
    {
        // Индекс наиболее подходящего цвета
        int curr_index = -1;
        // Расстояние в пространстве цветов
        long curr_error = INT_MAX;
```

```
        for (int color_index = 0; color_index < 256;
             color_index++)
```

```
        {
            // Нахождение значений r,g,b,
            // соответствующих переменной rgbindex;
            // предполагаемый формат 5.6.5. Затем
            // преобразование в формат 8.8.8, поскольку
            // палитра всегда задается в этом формате
            int r = (rgbindex >> 11) << 3;
            int g = ((rgbindex >> 5) & 0x3f) << 2;
            int b = (rgbindex & 0x1f) << 3;
```

```
            // Вычисление расстояния от исходного цвета
            // до текущего
            long delta_red =
                abs(src_palette[color_index].peRed - r);
            Long delta_green =
                abs(src_palette[color_index].peGreen - g);
            Long delta_blue =
                abs(src_palette[color_index].peBlue - b);
            Long error =
                (delta_red*delta_red)+
                (delta_green*delta_green)+
                (delta_blue*delta_blue);
```

```
            // Подходит ли этот цвет лучше других?
```

```

        if (error < curr_error)
        {
            currindex = color_index;
            curr_error = error;
        } // if

    } // for color_index

    // Найден наиболее подходящий цвет,
    // заносим его в таблицу
    rgblookup[rgbindex] = currindex;

} // for rgbindex

} // if
else
    if (rgb_format==DD_PIXEL_FORMAT555)
    {
        // Все элементы занимают 32 Кбайта памяти.
        // Создадим цикл для их поиска. Сведем объем
        // работы к минимуму, что важно даже для
        // процессора Pentium. Всего здесь 32768*256
        // итераций!
        for (int rgbindex = 0; rgbindex < 32768;
            rgbindex++)
        {
            // Индекс наиболее подходящего цвета
            int currindex = -1;
            // Расстояние в пространстве цветов
            long curr_error = INT_MAX;

            for (int color_index = 0; color_index < 256;
                color_index++)
            {
                // Нахождение значений r,g,b,
                // соответствующих переменной rgbindex;
                // предполагаемый формат 5.6.5. Затем
                // преобразование в формат 8.8.8,
                // поскольку палитра всегда задается в
                // этом формате
                int r = (rgbindex >> 10) << 3;
                int g = ((rgbindex >> 5) & 0x1f) << 3;
                int b = (rgbindex & 0x1f) << 3;

                // Вычисление расстояния от исходного
                // цвета до текущего
                long delta_red =
                    abs(src_palette[color_index].peRed-r);
                long delta_green =
                    abs(src_palette[color_index].peGreen-g);
                long delta_blue =
                    abs(src_palette [color_index].peBlue-b);
                Long error = (delta_red*delta_red) +

```

```

        (delta_green*delta_green) +
        (delta_blue*delta_blue);

    // Подходит ли этот цвет лучше других?
    if (error < curr_error)
    {
        curr_index = color_index;
        curr_error = error;
    } // if
} // for color_index

// Найден наиболее подходящий цвет,
// заносим его в таблицу
rgblookup[rgbindex] = currindex;

} // for rgbindex
} // if
else
    return(-1); // Серьезная проблема! Формат,
                // который не поддерживается

// Успешное завершение
return(1);

} // RGB_16_8_IndexedRGB_Table_Builder

```

Приведенная выше функция работает именно так, как было указано. Известно, что для 16-битового формата цвета входное значение находится в интервале от 0 до 65535. Таким образом, с помощью функции просто создается таблица соответствия путем вычисления индекса цвета, который подходит лучше всех остальных. Эту функцию следует вызывать после инициализации DirectDraw и загрузки 8-битовой палитры. Это делается следующим образом.

```

// Выделение памяти для таблицы соответствия,
UCHAR rgblookup - (UCHAR *)malloc(65536);

// При вызове используется глобальная
// битовая глубина и палитра
RGB_16_8_IndexedRGB_Table_Builder(dd_pixel_format, palette,
    rgblookuptable);

```

По окончании работы этой функции массив rgblookup[] будет содержать таблицу преобразования из 16-битового режима 5.6.5 или 5.5.5 в 8-битовый формат. Таким образом, мы вычисляем интенсивность освещения обычным путем, преобразуем полученный пиксель в 16-битовый формат, а затем выводим его на экран с помощью следующей инструкции.

```
final_pixel_index = rgblookup[final_pixelrgb16];
```

Вот и все! Посмотрим, что же мы здесь получили. Добавив предварительный шаг, мы полностью свели все преобразования к единой таблице соответствия, с помощью которой цвет каждого пикселя переводится в 8-битовый режим. Вопрос в том, нужно ли это? Ответ заключается в том, что избавиться от этого шага не удастся никаким образом. Если освещение моделируется в 8-битовом режиме, результаты в любом случае тем или иным путем придется преобразовывать в 8-битовые индексы. Конечно же, можно прибегнуть к трюкам. Например, создать палитру цветов с оттенками для каждого цвета, после чего данная ин-

тенсивность может использоваться в качестве индекса, нумеруя цвета в таблице, начиная от базового. Однако в данном методе тоже, по сути, используется индекс. Вывод, который следует из этих рассуждений, заключается в том, что моделирование освещения выполняется в пространстве RGB, а затем результат преобразуется в 8-битовые индексы.

Вопрос в том, можно ли создать цветовую модель, которая была бы в большей степени ориентирована на 8-битовый формат, чтобы свести к минимуму сложность преобразования из 16-битового формата RGB? Конечно же, можно. Посмотрим, как это делается.

## Упрощенная модель интенсивности освещения для 8-битовых режимов

Наиболее трудоемкий этап моделирования затенения в любом режиме — вычисление конечного RGB-значения. Однако если модель освещения упростить таким образом, чтобы не пришлось беспокоиться о цвете, мы получим возможность пользоваться предельно простой 8-битовой моделью затенения. Вот как это работает: все многоугольники обладают в пространстве RGB теми цветами, которые задаются палитрой. А теперь, поскольку мы имеем дело с 8-битовыми индексами и палитрой в формате 8.8.8, забудем на время обо всем, что связано с форматом RGB, продумаем весь процесс до конца и смоделируем освещение, продвигаясь в обратном направлении. Другими словами, нужно вычислить все 256 возможных значений интенсивности цвета, исходя из имеющейся в наличии палитры. После этого создадим таблицу с 256 строками, каждая из которых отвечает своему цвету реальной палитры. Все строки таблицы содержат по 256 элементов, которые представляют собой индексы палитры, наиболее соответствующие 256 значениям интенсивности данного цвета. Структура описанной таблицы представлена на рис. 8.26.

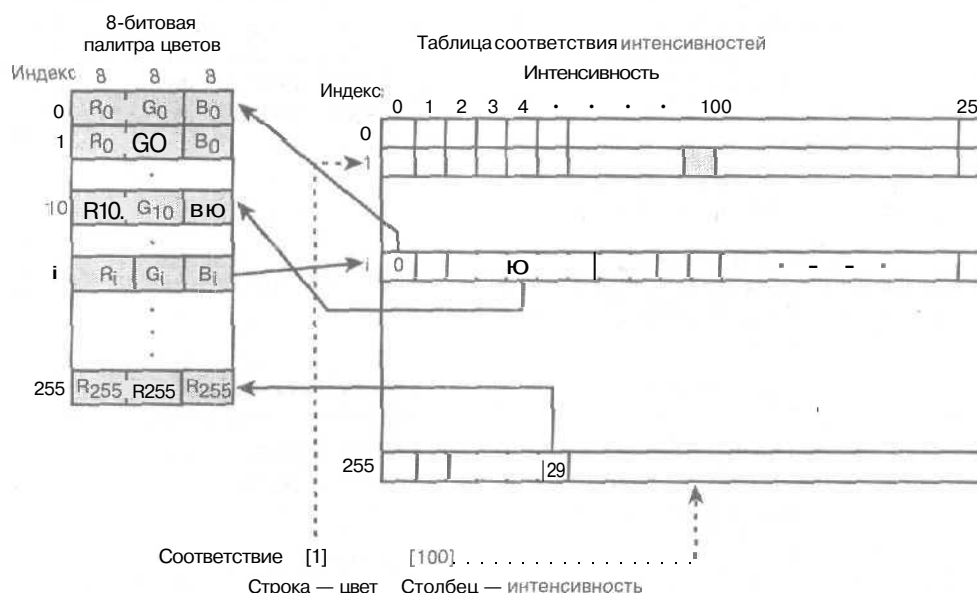


Рис. 8.26. Структура таблицы преобразования интенсивности индексированных цветов

Вопрос в том, зачем вообще вычислять значения общей интенсивности. Вычислим интенсивности только существующих цветов. Затем, на этапе моделирования освещения, этот цвет будет использоваться как индекс строки, а его интенсивность — как индекс

столбца таблицы. Как и раньше, мы пользуемся таблицей соответствия, однако всех промежуточных шагов, связанных с 16-битовым форматом RGB, удалось избежать. Это достигается путем интенсивной эксплуатации 8-битовой модели и тем, что значения интенсивности ограничены интервалом (0..255). Даже если на самом деле интенсивности занимают более широкий интервал, это не страшно, поскольку всегда можно сделать масштабирующее преобразование.

Если приведенные выше рассуждения сбили вас с толку, попробуем рассмотреть вопрос в другом аспекте. В первой, более упрощенной модели использовалось все пространство RGB. Другими словами, вычисления цвета и интенсивности производились в пространстве RGB, затем полученный 16-битовый цвет в формате RGB преобразовывался с помощью таблицы соответствия. Однако мы знаем, что уже не находимся в пространстве RGB. Так зачем же все усложнять? Конечно же, первый подход является более общим. Однако давайте посмотрим на все с точки зрения того, кто пытается ответить на вопрос: какие оттенки цвета в палитре играют главную роль? Тогда для каждого содержащегося в палитре цвета можно просто создать таблицу преобразования интенсивностей, в которой будет содержаться 256 **интенсивностей** каждого цвета. Это работает по той простой причине, что нам заранее известно, что модели цветов в такой 8-битовой имитации всегда будут совпадать с одним из цветов палитры! Поскольку мы искусственно организуем поиск по 256 оттенкам каждого цвета, то цвета с ограниченным количеством оттенков, конечно же, выглядят не **лучшим** образом, но это не столь важно.

Я кратко ознакомлю вас с функцией, которая выполняет поиск по описанной схеме. Однако в процессе ее использования следует помнить о том, что если мы хотим использовать оба описанных метода, то нужно разработать два различных препроцессора для вычисления **оттенков** в 8-битовых режимах. Один из них будет производить вычисления в пространстве RGB, а затем в последнюю минуту с помощью функции `rgblookup[]` будет осуществляться переход к 8-битовым индексам. При этом цвета в формате RGB будут представлены в виде индексов. Однако в новой модели будет использоваться двумерная таблица соответствия, в которой в качестве первого индекса будет выступать цвет, а в качестве второго — интенсивность: `rgbintensitylookup[color][intensity]`.

Чтобы создать **функцию**, генерирующую эту таблицу, понадобится создать цикл по всем цветам палитры, а затем вычислить все 256 оттенков каждого цвета в формате RGB. При этом вот в чем следует разобраться; нужно ли при вычислении 256 интенсивностей принимать интенсивность данного цвета за 100%, а затем двигаться вниз по **шкале**? **Поясним** суть проблемы на примере. Пусть мы пытаемся вычислить все интенсивности цвета с индексом 5, значения RGB-каналов которого равны (100,40,30). Проблема в том, что модель RGB не лучшим образом подходит для этого, поскольку действительно **понадобится** создать цвет, а затем — **его** 256 интенсивностей.

Приведем еще один пример. Пусть у нас есть цвет со значением (100,0,0), представляющий собой красный цвет половинной **интенсивности**. Следует ли при создании 256 оттенков этого цвета принимать данную интенсивность в качестве максимальной? Или же с помощью специальных математических преобразований нужно вычислить виртуальный цвет (255,0,0), а затем создать все его оттенки? По-видимому, такой подход в большей степени соответствовал бы ожиданию пользователя и дал бы лучшие **результаты** при работе **процессора** для работы с освещением.

Эти вопросы действительно важны, однако в настоящий момент мы будем следовать такому соглашению. Для каждого индекса будет **осуществляться** поиск RGB-значений **соответствующего** цвета, а затем интенсивность данного цвета будет увеличиваться до максимально возможного значения, которое **определяется** условием, чтобы ни в одном

из каналов не было переполнения. После этого, принимая новый **цвет** с увеличенной яркостью в качестве **исходного**, вычислим 256 **интенсивностей** данного цвета и произведем по ним поиск. Этот метод по своему эффекту аналогичен моделям затемнения, однако в нем используется более **однородное** пространство цветов. Приведем пример вычисления для данного исходного цвета,  $c$ , цвета  $c^*$  с максимальной интенсивностью.

#### Алгоритм модулирования интенсивности

Дано: многоугольник цвета  $c = \text{RGB}(r, g, b) = \text{RGB}(50, 20, 100)$ ;  $c^* = \text{RGB}(r^*, g^*, b^*)$ .

Шаг 1. Находим **компонент** с максимальным значением ( $b = 100$ ).

Шаг 2. Приводим компонент с максимальным значением к значению 255, вычисляя коэффициент **преобразования**:

$c = 100$ ,  $c^* = 255$ ,  $k = 255/100 = 2.55$ .

Шаг 3. **Преобразуем** остальные цвета с использованием найденного коэффициента:

$r^* = r \cdot k = 50 \cdot 2.55 = 127.5$ ,

$g^* = g \cdot k = 20 \cdot 2.55 = 51.0$ ,

$b^* = 255$ .

После округления получаем:  $c^* = \text{RGB}(128, 51, 255)$ . Цвета  $c^*$  и  $c$  — это один и тот же цвет разной интенсивности, поскольку отношение R:G:B в них одинаковое. Теперь примем интенсивность цвета  $c$  данным индексом за 100% и вычислим 256 значений интенсивности. Описанный процесс очень похож на тот, который выполнялся предыдущей функцией. Однако на этот раз вместо того, чтобы находить **общие** значения RGB, мы собираемся сгенерировать их и провести поиск. Данный алгоритм работает следующим образом: для каждого цвета выполняется преобразование его интенсивности до максимально возможной, затем полученная интенсивность принимается за 100% и на ее основе находят 256 новых значений интенсивности в нисходящем порядке, пока интенсивность в каждом канале не упадет до 0. По каждому сгенерированному таким образом значению RGB затем выполняется поиск в реальной **палитре** цветов по методу наименьших квадратов. После нахождения цвета, который подходит больше **всего**, он **заносятся** в столбец с индексом цвета, с которым мы работаем. Например, если у нас есть цвет  $25 \rightarrow \text{RGB}(255, 0, 0)$ , то поиск должен производиться по цветам  $\text{RGB}(0, 0, 0)$ ,  $\text{RGB}(1, 0, 0)$ , ...,  $\text{RGB}(255, 0, 0)$ .

Я надеюсь, что не слишком долго толоч воду в ступе, объясняя очевидные **вещи**, касающиеся 8-битовых режимов. Эти вопросы являются довольно тонкими и очень полезными в программных процессорах. Поэтому я хотел бы, чтобы вы усвоили все премудрости преобразования **цветов** в процессе программирования игры,

Как и было обещано, ниже приведена функция, которая создает таблицу **интенсивностей** для данной палитры. Обратите внимание, что битовая глубина не принимает участия в вычислениях, поскольку они выполняются в пространстве RGB 8.8.8, а результаты представляют собой индексы 8-битовой палитры. Таким образом, для работы функции понадобится только **входная** палитра и место в памяти для хранения таблицы (она будет занимать 64 Кбайт — 256 строк на 256 столбцов, по 1 байту на каждую ячейку).

```
int RGB_16_8_Indexed_Intensity_Table_Builder
(LPpaletteENTRY src_palette,
 UCHAR rgblookup[256][256], // Таблица соответствия
 int intensity_normalization=1)
{
    // В данной функции на основе исходной палитры
```

```

// вычисляются элементы таблицы градации
// интенсивностей; каждой строке таблицы отвечает
// индекс цвета, а каждому столбцу - интенсивность
// в интервале 0..255. Функция возвращает однобайтовый
// индекс. В вызывающей программе необходимо выделить
// буфер памяти объемом 64 Кбайта для [256] [256]
// однобайтовых элементов таблицы rgblookup, поскольку
// в функции память не выделяется. Для построения
// таблицы в функции организован цикл по всем цветам
// палитры. Интенсивность каждого цвета повышается до
// максимально возможной (не допуская при этом
// переполнения каналов RGB), а затем она принимается за
// 100% интенсивности для данного цвета, после чего
// находятся 256 других его интенсивностей в нисходящем
// порядке. Далее, по методу наименьших квадратов
// находится наиболее подходящий цвет палитры, который
// заносится в столбец, соответствующий тому цвету, для
// которого выполнялось преобразование интенсивностей.
// Примечание: если нормировочный коэффициент
// интенсивности задается равным 0, то повышение яркости
// цвета до максимальной не выполняется
int ri,gi,bi; // Начальный цвет
int rw,gw,bw; // Текущий цвет
float ratio; // Коэффициент преобразования
float dL,dr,db,dg; // Градиенты интенсивности для 256
// оттенков

// Сначала проверяем указатели
if (!src_palette || !rgblookup)
    return (-1);
// Для каждого цвета палитры вычисляем максимальное
// значение интенсивности, а затем находим 256 оттенков
for (int col_index = 0; col_index < 256; col_index++)
{
    // Извлечение цвета из палитры
    ri = src_palette[col_index].peRed;
    gi = src_palette[col_index].peGreen;
    bi = src_palette[col_index].peBlue;

    // Нахождение канала с наибольшей интенсивностью,
    // присвоение ему максимального значения (255), а
    // затем преобразование других каналов на основе
    // найденного коэффициента перехода
    if (intensity_normalization==1)
    {
        // Красный канал самый интенсивный?
        if (ri >= gi && ri >= bi)
        {
            // Вычисляется коэффициент перехода
            ratio = (float)255/(float)ri;

            // Цвет с максимальной интенсивностью
            ri = 255;

```

```

    gi = (int)((float)gi * ratio + 0.5);
    bi = (int)((float)bi * ratio + 0.5);
} // if
else // Зеленый канал самый интенсивный?
    if (gi >= ri && gi >= bi)
    {
        // Вычисляется коэффициент перехода
        ratio = (float)255/(float)gi;

        // Цвет с максимальной интенсивностью
        gi = 255;
        ri = (int)((float)ri * ratio + 0.5);
        bi = (int)((float)bi * ratio + 0.5);
    } // if
else // Самый интенсивный - синий канал
{
    // Вычисление коэффициента перехода
    ratio = (float)255/(float)bi;
    // Цвет с максимальной интенсивностью
    bi = 255;
    ri = (int)((float)ri * ratio + 0.5);
    gi = (int)((float)gi * ratio + 0.5);
} // if

} // if

// На данном этапе нужно вычислить градиенты
// интенсивности для текущего цвета, поэтому
// вычислим RGB значения для его 256 интенсивностей
dl = sqrt(ri*ri + gi*gi + bi*bi)/(float)256;
dr = ri/dl;
db = gi/dl;
dg = bi/dl;

// Инициализация рабочего цвета
rw = 0;
gw = 0;
bw = 0;

// На данном этапе rw,gw,bw - это цвет, для
// которого нужно вычислить 256 интенсивностей,
// чтобы ввести их в столбец таблицы с
// индексом col_index
for (int intensity_index = 0; intensity_index < 256;
    intensity_index++)
{
    // Индекс наиболее подходящего текущего цвета
    int curr_index = -1;
    // Расстояние в пространстве цветов до наиболее
    // подходящего цвета.
    long curr_error = INT_MAX;
    for (int color_index = 0; color_index < 256;
        color_index++)

```

```

i
// Расстояние в пространстве цветов
long delta_red =
abs(src_palette[color_index].peRed - rw);
long delta_green =
abs(src_palette[color_index].peGreen - gw);
long delta_blue =
abs(src_palette[color_index].peBlue - bw);
long error = (delta_red*delta_red) +
(delta_green*delta_green) +
(delta_blue*delta_blue);
// Подходит ли данный цвет лучше других?
if (error < curr_error)
{
curr_index = color_index;
curr_error = error;
} // if

} // for color_index

// Найден наиболее подходящий цвет,
// заносим его в таблицу
rgblookup[col_index][intensity_index]
= curr_index;

// Вычисление очередного уровня интенсивности с
// проверкой на переполнение (его быть не должно,
// но кто знает...)
if (rw+dr > 255) rw=255;
if (gw+dg > 255) gw=255;
if (bw+db > 255) bw=255;

} // for intensity_index
} // for c_index
// Успешное завершение
return(l);

} // RGB_16_8_Indexed_Intensity_Table_Builder

```

Для создания таблицы достаточно вызвать функцию таким же образом, как это делалось для ее RGB-версии.

```

UCHAR rgblookup[256][256];
RGB_16_8_Indexed_Intensity_Table_Builder(palette,
rgblookup, 1);

```

Приведем пример использования таблицы. Допустим, у нас есть многоугольник, индекс цвета для которого равен 12. Какой на самом деле **цвет** соответствует этому индексу, нас волнует меньше всего. Однако нужно, чтобы конечным цветом, соответствующим индексу 12, был цвет с интенсивностью 150 (максимальной интенсивности, конечно же, соответствует значение 255).

```
Final_pixel = fgblookup[12][15];
```

Вот как все просто!

Если в функции `intensity_normalization` последний параметр сделать равным 0, тем самым будет пропущен шаг, связанный с нормировкой. Если при этом в функции, например, обрабатывается цвет `RGB(100,10,10)`, то он берется в качестве исходного цвета, на основании которого вычисляются 256 оттенков. В противном случае (когда упомянутый параметр отличен от нуля) эти 256 оттенков вычисляются на основе цвета, яркость которого повышена по эвристическому алгоритму, в результате чего мы получаем цвет `RGB(255,25,25)`.

## Постоянное затенение

Итак, пора перейти к практике! Приступим к созданию функций, необходимых для работы игры — в 8-битовом режиме на медленных компьютерах и 16-битовом на машинах Pentium 4+. Первым видом затенения, который будет рассмотрен, является постоянное затенение. Его моделирование отличается предельной простотой, поскольку ничего не нужно учитывать — ни источников общего, ни источников диффузного света. Просто нужно сделать так, чтобы многоугольники были заполненными и имели определенный цвет. Таким образом, при моделировании постоянного затенения нас не должны волновать источники света. Мы просто берем индекс или RGB-значение цвета многоугольника и выводим его на экран без изменений. Все происходит так, как если бы мы моделировали многоугольники, излучающие свет определенного цвета и определенной интенсивности, т.е. светящиеся многоугольники. При их выводе на экран используется тот цвет, который задан для них изначально. Конечно же, как уже отмечалось, такие "излучающие" многоугольники на самом деле не испускают свет, однако *все выглядит именно так*. Если бы мы хотели, чтобы предметы действительно излучали свет, их нужно было бы отнести к источникам,

Теперь у нас есть все, что нужно для того, чтобы прямо сейчас продемонстрировать пример постоянного затенения. Я собираюсь воспользоваться имитатором танковой битвы, уже встречавшемся нам в главе 7, и заменить вызов функции, которая выводит многоугольники в виде каркасов, разработанной в данной главе функцией, выводящей заполненные многоугольники. Старый код (для 16-битового режима) выглядел следующим образом.

```
// Вывод объекта
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer,
                             back_lpitch);
```

А теперь приведем новый вызов.

```
// Вывод объекта
Draw_RENDERLIST4DV1_Solid16(&rend_list, back_buffer,
                              back_lpitch);
```

Для каждого объекта в этом демонстрационном примере (танков, прицела, башен) мне пришлось сгенерировать новые файлы в формате .PLG. Это нужно для того, чтобы обеспечить правильность поворота каждого многоугольника, а также правильность флагов, указывающих односторонность и двусторонность. Этот шаг необходим, поскольку теперь мы выводим заполненные объекты и хотим иметь гарантию того, что обратные поверхности многоугольников не обрабатываются и не выводятся.

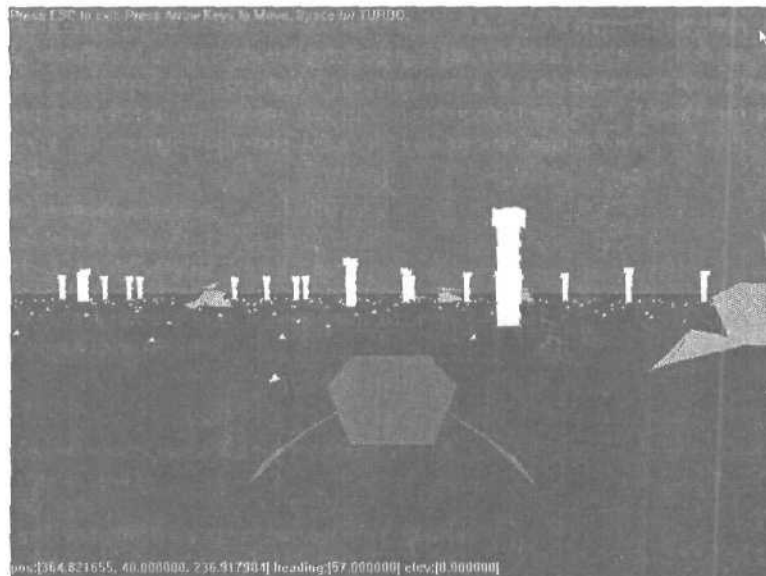


Рис. 8.27. Экранный снимок, полученный при работе первой демонстрационной программы, предназначенной для моделирования затенения

На рис. 8.27 показан вывод программы, который мы увидим после указанной выше замены вызываемой функции. Исходный и исполняемый файлы содержатся в файлах DEMOII8\_4.CPP|EXE (DEMOII8\_4\_8b.CPP|EXE). А теперь кратко рассмотрим фрагмент, отвечающий за поддержку формата PLX/PLG.

```
// Флаг двусторонности
#define PLX_2SIDED_FLAG      0x1000
    // Двусторонний многоугольник.
#define PLX_1SIDED_FLAG      0x0000
    // Односторонний многоугольник.

// Режим затенения многоугольников
#define PLX_SHADE_MODE_JURELAG 0x0000
    // Многоугольник обладает постоянным цветом
#define PLX_SHADE_MODE_CONSTANT_FLAG 0x0000
    // Псевдоним предыдущего макроса
#define PLX_SHADE_MODE_FLAT_FLAG 0x2000
    // Для данного многоугольника используется
    // плоскостное затенение
#define PLX_SHADE_MODE_GOURAUD_FLAG 0x4000
    // Для данного многоугольника используется
    // затенение по Гуро
#define PLX_SHADE_MODE_PHONG_FLAG 0x6000
    // Для данного многоугольника используется
    // затенение по Фонгу
#define PLX_SHADE_MODE_FASTPHONG_FLAG 0x8000
    // Для данного многоугольника используется
    // затенение по Фонгу (альтернативный подход)
```

На данном этапе в функциях `Draw_RENDERLIST4DV1*()` модель затенения не учитывается. Однако формат `PLG/PLX` поддерживает различные виды затенения — какие мы только пожелаем. В данном случае это можно было бы сделать с помощью флага `PLX_SHADE_MODE_CONSTANT_FLAG`, но проблема в том, что функция даже не обратит на него внимания. Код, который будет разработан в следующей главе, будет способен работать с этими флагами, поскольку мы собираемся организовать полную поддержку всех концепций модели освещения и соответствующих функций. Однако сейчас я не собираюсь переписывать весь игровой процессор. Было описано так много новых идей, что я просто хочу реализовать их на том уровне, на котором мы находимся сейчас, а в следующей главе мы все усовершенствуем.

Доработаем демонстрационный пример с учетом сказанного, а затем через некоторое время вернемся к этому вопросу. Хотелось бы, чтобы вы обратили внимание на одно обстоятельство... Рассматривая данный демонстрационный пример или внимательно читая эту книгу, можно было бы заметить один серьезный недостаток визуализации: многоугольники, которые выводятся на экран, неупорядочены! Другими словами, нам нужно реализовать некоторую сортировку по глубине, чтобы знать, какие многоугольники расположены дальше от наблюдателя, а какие — ближе. Один из методов, с помощью которого можно устранить этот недостаток, — алгоритм художника (*painter's algorithm*). Этот алгоритм прост в реализации, и если геометрия несложная, он способен справиться с сортировкой многоугольников и правильно вывести изображение. Пока что я не хочу реализовывать этот алгоритм, поскольку это вопрос видимости предметов (хотя к концу главы он будет реализован). Этот алгоритм будет применяться до тех пор, пока мы не перейдем к более сложным алгоритмам, отвечающим за видимость предметов и работу с буфером глубины. В основе алгоритма, о котором идет речь, лежит простая идея. Пусть у нас есть список визуализации. Далее многоугольники сортируются по средним значениям координаты *z*, причем сначала идут более удаленные, а затем — те, которые расположены все ближе к наблюдателю. После этого они выводятся на экран именно в таком порядке. Это помогает решить большинство проблем, возникающих в нашем демонстрационном примере при формировании изображения на экране. Описанный процесс проиллюстрирован на рис. 8.28, однако этим мы займемся позже. А сейчас попробуем улучшить модель освещения.

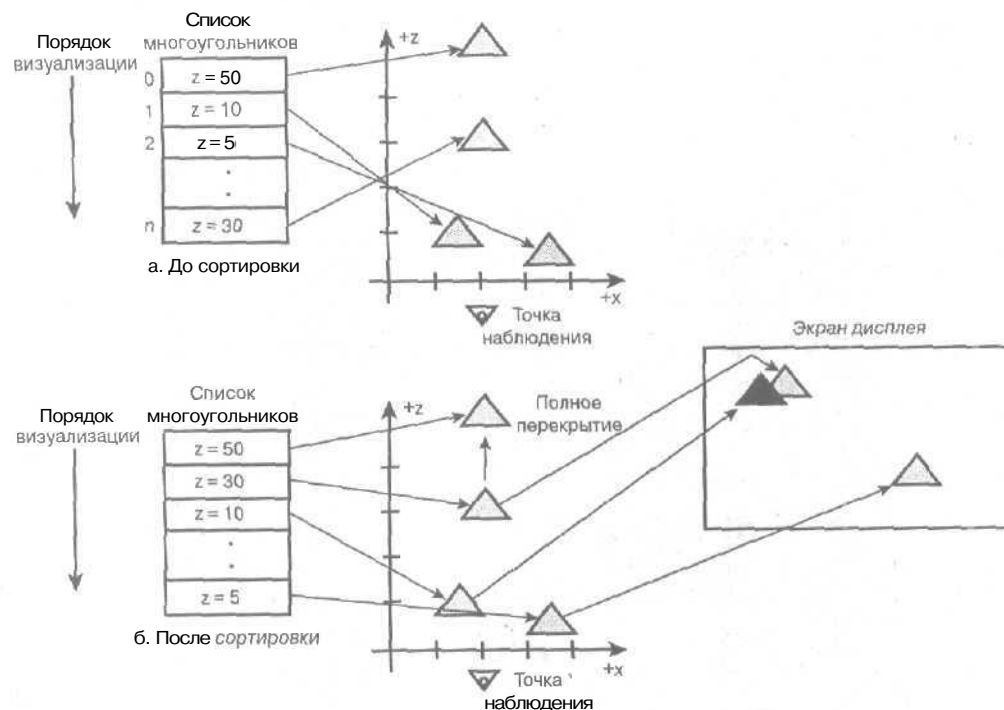


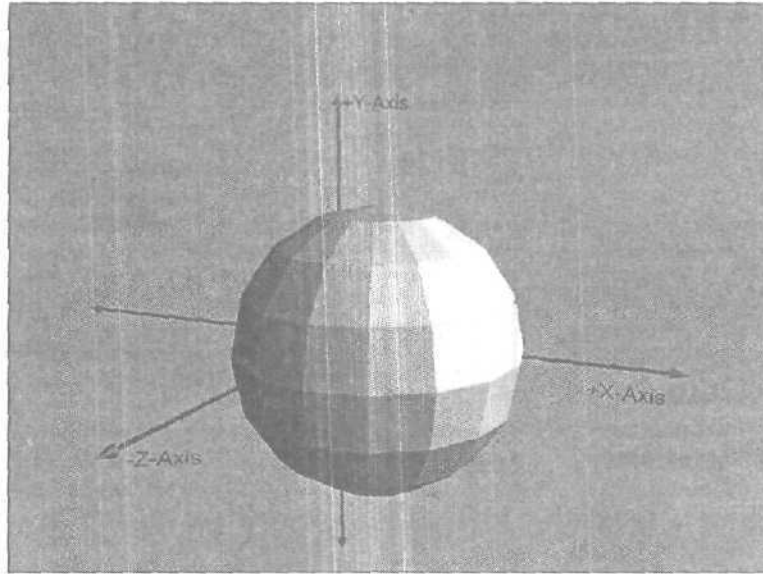
Рис. 8.28. Алгоритм затухивания и  $z$ -сортировка

## Плоское затенение

Итак, мы вплотную подошли к моделированию реалистичного освещения. *Плоское затенение* (flat shading) — это такое затенение, при котором для определения цвета некоторого многоугольника достаточно найти цвет одного его пикселя. Другими словами, мы предполагаем, что многоугольник изготовлен из единого материала, а поскольку все многоугольники в нашем процессоре являются треугольниками, очевидно, что это плоский многоугольник, так что вектор нормали одинаков в любой точке его поверхности. Таким образом, вычисления, связанные с освещением, можно выполнить для одной из вершин каждого многоугольника, а затем затенять весь многоугольник, основываясь на этих результатах. Это и есть плоскостное или *фасетное* (faceted) затенение. Для объектов, которые по своей природе состоят из плоских поверхностей, все выглядит хорошо, однако если поверхность объекта искривленная, она приближенно представляется в виде набора многоугольников. Поэтому изображение объекта состоит как бы из множества граней. Описанный эффект проиллюстрирован на рис. 8.29. Если же применить методы сглаживания затенения, например, затенение по Гуро или по Фонгу, то в такой модели многогранники с острыми гранями будут выглядеть так, как будто эти грани закруглены. Мы еще вернемся к этому вопросу, а пока что попробуем найти путь для реализации плоского затенения.

Посмотрим, как это делается. У нас есть загрузчик файлов в формате PLG/PLX, поэтому можно добавить в наши простые модели возможность задавать тип затенения и свойства цвета поверхности (в формате RGB или в виде индекса цвета). Все это хорошо, однако у нас пока что нет ни одной функции для моделирования освещения, по-

этому нужно об этом позаботиться. Как уже было сказано, до тех пор, пока **освещение** моделируется в глобальной системе координат или в системе, связанной с камерой, у нас не должно возникать проблем. Вопрос в том, на каком уровне следует **осуществлять** поддержку **освещения**: на уровне объектов, на уровне списка визуализации или на обоих уровнях?



*Рис. 8.29. Если объект приближенно представляется в виде набора многоугольников, он выглядит так, как будто состоит из лоскутков*

Первое, что нужно сделать, — найти в последовательности операций, которые выполняются при программировании игры, место, где можно заняться моделированием **освещения**. В качестве примера приведем несколько измененную последовательность, которая использовалась для разработки предыдущей демонстрационной программы. В представленном ниже фрагменте удален лишний код и выполнено определенное **обобщение**.

```
// Вывод списка визуализации
Reset_RENDERLIST4DV1(&rend_list);

// Генерация матрицы камеры
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// Обнуление параметров объекта
Reset_OBJECT4DV1(&obj);

// Переход к глобальным координатам
Model_To_World_OBJECT4DV1(&obj, TRANSFORM_TRANS_ONLY);

// Попытка отбраковать объекты
Cull_OBJECT4DV1(&obj, &cam, CULL_OBJECT_XYZ_PLANES))

// Помещение объекта в список визуализации
Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj);
```

```

// Удаление обратных поверхностей
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);

// Переход от глобальной системы отсчета
// к системе отсчета камеры
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// Проецирование в системе отсчета камеры
// на плоскость экрана
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// Переход к системе отсчета экрана
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// Визуализация объекта
Draw_RENDERLIST4DV1_Solid16(&rend_list, back_buffer,
                             back_lpitch);

```

В данном примере в список визуализации помещается только один объект `obj`. На этом пути возникает много вопросов, которые следовало бы продумать. Кратко опишем процесс. Сначала игровой процессор проверяет, не подлежит ли данный объект отбраковке. Если объект не отбраковывается, он попадает в список визуализации, после чего в нем удаляются элементы поверхности, нормаль к которым направлена от наблюдателя, и производится ряд преобразований системы отсчета. Первым осуществляется переход от глобальной системы отсчета к системе отсчета, связанной с камерой. Затем производится проецирование на плоскость экрана, и **наконец**, преобразование, связанное с направлением осей координат монитора. Последней осуществляется визуализация с **помощью** функции `Draw_RENDERLIST4DV1_Solid16()`.

Технически моделирование освещения можно выполнить в любом месте, расположенном между вызовами функций `Model_To_World_OBJECT4DV1()` и `Camera_To_Perspective_RENDERLIST4DV1()`. Здесь возникает дилемма. Есть несколько подходящих мест, куда можно поместить код, отвечающий за освещение. Например, этот код можно сделать частью функции, с помощью которой формируется объект. Это означает, что можно **создать** отдельную функцию, с помощью которой освещается сам объект, и вызвать ее после того, как отбракованы объекты, находящиеся вне поля зрения (при этом не придется тратить время на освещение объектов, которые не будут выводиться на экран). С другой стороны, имитирующий **освещение** код можно было бы добавить в функцию, в которой удаляются обратные поверхности объектов. Причиной для этого могло бы послужить то, что на данном этапе мы работаем с нормальными к поверхностям **многоугольников**, а нормали — это ключевые элементы в моделировании освещения. Однако **совмещение** удаления обратных поверхностей и освещения в одной функции кому-то может показаться плохим стилем. Можно прибегнуть к такому трюку: на этапе удаления обратных поверхностей сохранить информацию о нормалях для того, чтобы использовать ее на следующем этапе, которым является моделирование освещения. Это всего лишь одна из нескольких возможностей. Словом, здесь есть над чем подумать.

На подобные вопросы не существует однозначных ответов. Единственно правильный путь — тот, который работает в данной конкретной ситуации. Поскольку эта книга является учебным пособием, мы разработаем для моделирования освещения две функции: одна из них выполняет освещение объекта на этапе его создания, а другая — после удаления обратных поверхностей объектов (в списках **визуализации**).

Пока что мы остановимся на том, что объединим в рамках одной функции удаление обратных поверхностей и **освещение** объекта. Причина этого в том, что при моделировании освещения выполняется так много вычислений, что не стоит писать код для дополнительных действий, связанных с сохранением информации о нормалях к элементам поверхности до тех пор, пока мы серьезно не **займемся** оптимизацией. Возможно, мы вернемся к этому позже, при описании оптимизации. А пока что просто займемся освещением объектов и списками визуализации.

**НА ЗАМЕТКУ**

Возможно, вы заметили, что я предпочитаю выполнять операции на уровне списка визуализации, а не на уровне самих объектов. Конечно же, причина заключается в том, что в списке визуализации содержится вся геометрия, с которой следует работать. Такой путь кажется более понятным, чем тот, при котором объекты освещаются по очереди, **один** за другим.

Теперь нам предстоит продумать детали. Например, если мы занимаемся написанием модуля, в котором освещение выполняется поочередно для каждого объекта, то нам понадобится изменять цвет каждого многоугольника. А достаточно ли у нас для этого места? Подобный вопрос возникает, если освещение выполняется на уровне списка визуализации. Наконец, если освещение моделируется в функции, в которой удаляются обратные поверхности, то достаточно ли у нас места, чтобы сохранить информацию о нормалях к поверхностям многоугольников?

Итак, начнем с решения первой задачи. Если освещается объект, который находится в пространстве игры, то где нужно хранить конечный цвет многоугольника? Исходный цвет перезаписывать нельзя, поскольку при следующем проходе будет утрачена исходная информация о многоугольнике. Помня об этом, рассмотрим структуру **OBJECT4DV1**.

```
// Объект создан на основе списка вершин
// и списка многоугольников
typedef struct OBJECT4DV1_TYP
{
    int id;           // Числовой идентификатор объекта
    char name[64];    // ASCII-имя объекта
    int state;        // Состояние объекта
    int attr;         // Атрибуты объекта
    float avg_radius; // Средний радиус объекта,
                    // использующийся при моделировании
                    // столкновений
    float max_radius; // Максимальный радиус объекта.
    POINT4D world_pos; // Положение объекта в глобальной
                    // системе координат
    VECTOR4D dir;     // Углы поворота объекта 8 локальных
                    // координатах или компоненты
                    // пользовательского единичного
                    // вектора, указывающего направление

    VECTOR4D ux,uy,uz; // Локальные оси для отслеживания
                    // общей ориентации (обновляются
                    // автоматически в процессе вызова
                    // функций, осуществляющих поворот
                    // объекта

    int num_vertices; // Количество вершин данного объекта
}
```

```

// Локальные вершины
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES];
// Координаты вершин после преобразования
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES];
// Количество многоугольников в объекте
int num_polys;
// Массив многоугольников
POLY4DV1 plist[OBJECT4DV1_MAX_POLYS];
} OBJECT4DV1, *OBJECT4DV1_PTR;

```

Сами многоугольники хранятся в массиве `plist[]`, каждый элемент которого представляет собой следующую структуру.

```

// Многоугольник, построенный на основе списка вершин.
typedef struct POLY4DV1_TYP
{
    int state; // Информация о состоянии
    int attr; // Физические атрибуты многоугольника
    int color; // Цвет многоугольника

    POINT4D_PTR vlist; // Список вершин
    int vert[3]; // Индексы в списке вершин
} POLY4DV1, *POLY4DV1_PTR;

```

К сожалению, нам не повезло! Оказывается, негде хранить цвет, полученный и результате затенения освещенного объекта. Но пока что я не хочу переписывать разрабатываемый игровой процессор. Сначала будет пройден этап доводки процессора до рабочего состояния и создания структур данных, чтобы в следующей главе на основе этой информации можно было создать новую версию. Итак, приступим к работе.

Информация о цвете хранится в 32-битовых переменных типа `int`, однако она занимает только 16 младших битов, поскольку поддерживаются 16- и 8-битовые модели цветов. Таким образом, информацию о цвете с учетом затенения можно поместить в старшие 16 битов. Однако это приведет к проблеме, поскольку нижние 16 битов используются функцией визуализации, которая моделирует затенение. Поэтому нужно слегка изменить код, отвечающий за вставку объекта (или переписать функцию визуализации). Эта функция должна считать, что многоугольник был освещен и его цвет уже внесен в старшие 16 битов. Затем функция должна использовать этот цвет при разложении объекта на многоугольники, заносимые в список визуализации. Таким образом, новая функция, отвечающая за вставку объекта в список визуализации, будет выглядеть следующим образом.

```

int Insert_OBJECT4DV1_RENDERLIST4DV2
(RENDERLIST4DV1_PTR rend_list,
 OBJECT4DV1_PTR obj,
 int insert_local = 0,
 int lighting_on = 0)
{
    // Преобразует объект в список его граней, затем
    // вставляет видимые, активные, неотсеченные и
    // неотбракованные многоугольники в список визуализации.
    // Флаг insert_local отвечает за использование списка
    // вершин vlist_local или vlist_trans. Присвоив
    // переменной insert_local значение 1 (значение по
    // умолчанию 0), можно поместить в список необработанный
    // объект, который не подвержен никаким преобразованиям.
}

```

```

// По умолчанию используется значение 0; при этом объект
// помещается в список визуализации, по крайней мере,
// после перехода из локальной системы координат в
// глобальную. Последний параметр указывает на то, был
// ли реализован этап освещения с генерацией цвета в
// старших 16 битах слова. Если lighting_on - 1, то при
// помещении многоугольника в список визуализации
// информация о его цвете берется из старших 16 битов

// Проверим, не является ли данный объект неактивным,
// отбракованным или невидимым.
if (!(obj->state & OBJECT4DV1_STATE_ACTIVE) ||
    (obj->state & OBJECT4DV1_STATE_CULLED) ||
    !(obj->state & OBJECT4DV1_STATE_VISIBLE))
    return(0);

// Объект подлежит визуализации; разберем его на
// отдельные многоугольники
for (int poly - 0; poly < obj->num_polys; poly++)
{
    // Запрос многоугольника
    POLY4DV1_PTR curr_poly = &obj->plist[poly];
    // Сначала проверим, является ли многоугольник
    // видимым
    if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
        (curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
        (curr_poly->state & POLY4DV1_STATE_BACKFACE))
        continue; // Переход к следующему многоугольнику

    // Перезапись списка вершин многоугольника;
    // предполагается, что используются его локальные
    // координаты
    // Сначала сохраним старый указатель
    POINT4D_PTR vlist_old = curr_poly->vlist;
    if (insert_local)
        curr_poly->vlist = obj->vlist_local;
    OR: else
        curr_poly->vlist = obj->vlist_trans;
    // Проверим, следует ли перезаписать цвет,
    // взяв его значение из старших 16 битов
    if (lighting_on)
    {
        // Сохранение цвета
        unsigned int base_color =
            (unsigned int)curr_poly->color;
        curr_poly->color = (int)(base_color >> 16);
    } // if

    // Вставка многоугольника
    if (!Insert_POLY4DV1_RENDERLIST4DV1(rend_list,
        curr_poly))

    // Установка указателя на список вершин
    curr_poly->vlist = vlist_old;
}

```

```

    // Весь объект не подходит!
    return(0);
} // if

// Проверим, следует ли перезаписать цвет,
// взяв его значение из старших 16 битов
if (lighting_on)
{
    curr_poly->color = (int)base_color;
} // if

// Установка указателя на список вершин
curr_poly->vlist = vlist_old;

} // for
// Успешное выполнение,
return(1);
} // Insert_OBJECT4DV1_RENDERLIST4DV2

```

Полужирным шрифтом выделен код, в котором происходит сохранение основного цвета и сдвига содержимого старших 16 битов в младшие 16 битов. Запомните, что этот прием работает как для 8-битовых, так и для 16-битовых режимов, поскольку нас меньше всего должны беспокоить значения, которые находятся в младших 16 битах. При занесении объекта в список визуализации их просто нужно будет восстановить. Если освещение уже реализовано, необходимо предположить, что в старших 16 битах содержится информация о цвете **освещенного** объекта в формате RGB или в виде индекса палитры. В любом случае мы просто сдвигаем верхние 16 битов в нижние 16 битов — и все.

Таким образом, в данный момент мы располагаем возможностью поочередно вставлять в список визуализации освещенные объекты (нам все еще нужна функция, в которой освещается объект), однако нужно учесть два других случая, в которых освещение объекта может выполняться в ходе удаления обратных элементов его поверхности, а также после удаления обратных поверхностей в списке визуализации. Здесь нам немного повезло. Во-первых, потому, что мы уже работаем со списком визуализации, в котором можно стирать цвет любого многоугольника (поскольку исходный цвет хранится в самих объектах). Таким образом, при разработке функции освещения можно не беспокоиться о проблеме перезаписи цвета.

Помимо вопросов, о которых шла речь выше, следует обдумать возможность объединения этапа удаления обратных поверхностей и этапа **освещения** в рамках одной функции. Для этого понадобилась бы создать новую функцию, но мы пока решили не делать этого, чтобы код оставался понятным. Вопрос в том, удастся ли смоделировать освещение в качестве отдельного этапа, используя при этом информацию о нормалях к элементам поверхности, полученную во время удаления обратных поверхностей? Другими словами, найдется ли в списке визуализации место, в котором можно было бы сохранить информацию о нормалях, чтобы впоследствии считывать ее отдельной функцией? Обе последовательности этапов визуализации, о которых идет речь, проиллюстрированы на рис. 8.30. Итак, взглянем на первую версию списков визуализации.

```

typedef struct RENDERLIST4DV1_TYP
{
    int state; // Состояние в списке визуализации
    int attr; // Атрибуты в списке визуализации

```

```

// Список визуализации - это массив указателей, каждый
// из которых указывает на самостоятельную
// визуализируемую поверхность многоугольника POLYF4DV1
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

// Чтобы избежать выделения и освобождения ресурсов в
// каждом кадре, будем хранить поверхности
// многоугольников в этом массиве
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys; // Количество многоугольников в списке
                // визуализации
} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;

```



Рис. 8.30. Различные варианты последовательностей этапов визуализации

Вскоре вы убедитесь, что список визуализации состоит не из простых многоугольников, а из их граней POLYF4DV1, которые выглядят следующим образом.

```

typedef struct POLYF4DV1_TYP
{
    int state; // Состояние информации.
    int attr; // Физические атрибуты многоугольника
    int color; // Цвет многоугольника.

    POINT4D vlist[3]; // Вершины данного треугольника
    POINT4D tvlist[3]; // Вершины после преобразования
                    // (при необходимости)
    POLYF4DV1_TYP *next; // Указатель на следующий
                        // многоугольник в списке
    POLYF4DV1_TYP *prev; // Указатель на предыдущий
                        // многоугольник в списке
} POLYF4DV1, *POLYF4DV1_PTR;

```

К сожалению, нормали сохранять негде: в каждом многоугольнике есть место для исходной информации о вершинах и для информации о вершинах после преобразования, и не более того. Итак, мы выяснили, что, несмотря на внешнюю привлекательность идеи о сохранении информации о нормалях в функции, выполняющей удаление обратных поверхностей (по крайней мере, в качестве альтернативы подходу, объединяющему удаление обратных поверхностей и освещение), у нас просто нет места для этой информации. Впрочем, можно было бы прибегнуть к хитрым трюкам и записывать эту информацию в указателях `next` и `prev`, однако для этого пришлось бы проделать очень много работы.

Вывод, к которому я хочу вас подвести, состоит в том, что определенные ранее структуры данных весьма недостаточны для наших целей. В этом нет ничего удивительного — ведь когда мы их задавали, то еще не совсем ясно представляли себе дальнейшие шаги!

## Реализация плоского затенения в 16-битовом режиме

Итак, нам предстоит разработать систему освещения объекта и помещения его в список визуализации. Пора подумать о том, источники освещения какого типа мы собираемся поддерживать. Выше было описано несколько видов источников, включая бесконечно удаленные источники и световые пятна. Кроме того, мы определились с тем, как задавать источники и материалы. Теперь нужно разработать соответствующий код как для 16-битового, так и для 8-битового режимов. Сначала рассмотрим 16-битовый случай. В качестве первой функциональной модели освещения мы используем одну из таких моделей:

- многоугольники, для которых установлен бит постоянного затенения, на этапе моделирования освещения не обрабатываются (т.е. остаются без изменений);
- многоугольники, для которых установлен бит плоского затенения, на этапе освещения подвергаются воздействию источников общего освещения, бесконечно удаленных и точечных источников, а также световых пятен; все указанные источники предварительно должны быть занесены в список источников освещения.

Принципы освещения не зависят от того, к чему они применяются, — к объектам или к элементам списка визуализации. Если освещение производится для элементов списка визуализации, можно не беспокоиться о сохранении цвета: информацию о цвете можно перезаписывать, поскольку после того, как содержимое списка визуализации выводится на экран, оно сбрасывается.

Далее нужно проверить атрибуты каждого многоугольника. Если для многоугольника установлен флаг `POLY4DV1_ATTR_SHADE_MODE_CONSTANT`, то освещение этого многоугольника не выполняется. Если же установлен флаг `POLY4DV1_ATTR_SHADE_MODE_FLAT`, соответствующий многоугольник освещается. Псевдокод для первой модели процессора освещения выглядит примерно так (см. также рис. 8.31):

1. вычисляем нормаль к элементу поверхности;
2. вычисляем интенсивность освещения от каждого источника из списка;
3. суммируем интенсивности;
4. записываем результаты в старшие 16 битов цвета многоугольника.

Конечно же, вычисления должны выполняться на основе моделей источников освещения, описанных в первой части настоящей главы. Кроме того, в нашей первой версии функции, моделирующей освещение, отсутствует концепция материалов. Таким образом, для всех объектов по умолчанию будет использован материал, в котором в роли коэффициентов отражения выступают сами цвета. Кроме того, объекты будут отражать только общий и рассеянный свет.

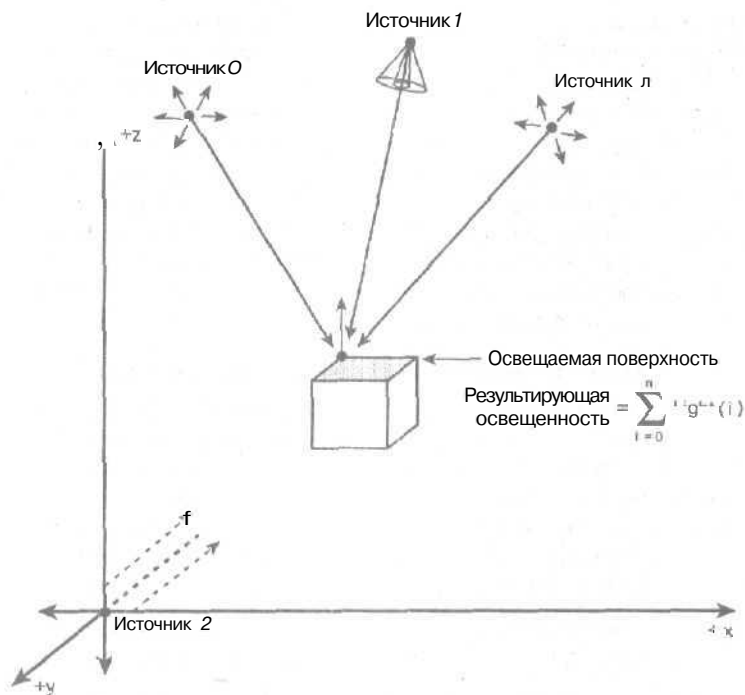


Рис. 8.31. Схема алгоритма освещения

Учитывая сделанные выше замечания, приведем код функции, осуществляющей освещение объекта (предполагается, что он задан в глобальных координатах). Функция разрабатывалась исходя из того, что в нее должны передаваться параметры источников освещения.

```
int Light_OBJECT4DV1_World16(
    OBJECT4DV1_PTR obj,      // Обрабатываемый объект
    CAM4DV1_PTR cam,        // Положение камеры.
    LIGHTV1_PTR lights,      // Список источников (может содержать
                             // несколько источников).
    int max_lights)          // Максимальное количество источников
                             // в списке
{
    // 16-битовая версия функции, моделирующей освещение
    // объекта на основе переданных в нее параметров
    // источников и камеры. Функция поддерживает постоянное
    // (излучательное) затенение, плоское затенение,
    // полученное в результате освещения объекта бесконечно
    // удаленными и точечными источниками, а также световыми
    // пятнами. Заметим, что в данной функции вычисления
    // выполняются довольно прямолинейно. Существуют более
    // эффективные операции с числами типа int но
    // целесообразность их применения под вопросом. Операции
    // над числами с плавающей запятой и над числами типа int
    // выполняются почти с одинаковой скоростью, однако для
    // преобразования чисел одного типа к другому потребуется
```

```

// большое количество операций. Если есть возможность
// выполнять вычисления с числами типа int, то вы
// выиграете в скорости. Помните о том, что световое пятно
// типа 1 - это просто точечный источник с выделенным
// направлением. Несмотря на наличие светового конуса, в
// котором интенсивность освещения спадает к его краю,
// такие источники все же выглядят как точечные,
// Интенсивность освещения от светового пятна типа 2
// пропорциональна косинусу угла между нормалью к элементу
// поверхности и направлением на источник. Величина  $\rho_f$ ,
// определяющая концентрированность интенсивности около
// оси светового конуса, должна принимать целочисленные
// значения: 1,2, 3,... и не должна быть дробной

unsigned int r_base, g_base, b_base, // Основной цвет
// освещения
r_sum, g_sum, b_sum, // Суммирование вкладов всех
// источников
shaded_color; // Конечный цвет

Float dp, // Скалярное произведение
dist // Расстояние от источника до поверхности
i, // Общие интенсивности
nL, // Длина нормали
atten; // Коэффициент затухания

// Проверяем, не отбракован ли объект
if (!(obj->state & OBJECT4DV1_STATE_ACTIVE) ||
(obj->state & OBJECT4DV1_STATE_CULLED) ||
!(obj->state & OBJECT4DV1_STATE_VISIBLE))
return(0);

// Цикл для обработки каждого многоугольника
for (int poly=0; poly < obj->num_polys; poly++)
{
// Извлечение параметров многоугольника
POLY4DV1_PTR curr_poly = &obj->plist[poly];

// Проверка многоугольника. Он тестируется только при
// условии, что он не выходит за область видимости, не
// отбраковывается, активный и видимый. Мы пытаемся не
// заниматься поиском обратных поверхностей, если они
// могли быть обнаружены на предыдущих этапах!
if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
(curr_poly->state & POLY4DV1_STATE_CLIPPED) ||
(curr_poly->state & POLY4DV1_STATE_BACKFACE))
continue; // Переход к следующему многоугольнику

// Извлечение индексов вершин в главный список
// Напомним, что многоугольники не являются
// самодостаточными элементами; их определение основано
// на списке вершин, хранящемся в самих объектах
int vindex_0 = curr_poly->vert[0];

```

```

int vindex_1 = curr_poly->vert[1];
int vindex_2 = curr_poly->vert[2];

// Будет использован список вершин преобразованного
// многоугольника, поскольку удаление обратных
// поверхностей имеет смысл лишь после перехода к
// глобальным координатам. Проверка режима затенения
// многоугольника (плоское затенение или затенение по
// Гуро)
if (curr_poly->attr & POLY4DV1_ATTR_SHADE_MODE_FLAT ||
    curr_poly->attr & POLY4DV1_ATTR_SHADE_MODE_GOURAUD)
{
    // Шаг 1: запрос базового цвета в режиме RGB
    if (dd_pixel_format == DD_PIXEL_FORMAT565)
    {
        _RGB565FROM16BIT(curr_poly->color, &r_base,
                          &g_base, &b_base);
        // Переход в 8-битовый режим
        r_base <<= 3;
        g_base <<= 2;
        b_base <<= 3;
    } // if
    else
    {
        RGB555FROM16BIT(curr_poly->color, &r_base,
                          &g_base, &b_base);
        // Переход в 8-битовый режим
        r_base <<= 3;
        g_base <<= 3;
        b_base <<= 3;
    } // if

    // Инициализация итогового цвета
    r_sum = 0;
    g_sum = 0;
    b_sum = 0;

    // Цикл по всем источникам
    for (int curr_light = 0; curr_light < max_lights;
         curr_light++)
    {
        // Активен ли этот источник?
        if (!lights[curr_light].state)
            continue;
        // Определение вида источника
        if (lights[curr_light].attr & LIGHTV1_ATTR_AMBIENT)
        {
            // Умножаем интенсивность в каждом канале на
            // соответствующую интенсивность цвета
            // многоугольника, а затем делим на 256, чтобы
            // результат был в интервале 0..255. В реальной
            // ситуации пользуйтесь побитовым сдвигом (>>8)
            r_sum += ((lights[curr_light].c_ambient.r *

```

```

    r_base) / 256);
g_sum+= ((lights[curr_light].c_ambient.g *
    g_base) / 256);
b_sum+= ((lights[curr_light].c_ambient.b *
    b_base) / 256);
// Лучше задавать только один источник общего
// света!
} // if
else if (lights[curr_light].attr &
    LIGHTV1_ATTR_INFINITE)
{
    // Для бесконечно удаленного источника нужна
    // нормаль к поверхности и направление на источник

    // Нужно вычислить нормаль к поверхности
    // многоугольника; напомним, что вершины задаются
    // по часовой стрелке, u=p0->p1, v=p0->p2, n=uxv
    VECTOR4D u, v, n;

    // Построение векторов u, v
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_1 ], &u);
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_2 ], &v);

    // Вычисление векторного произведения
    VECTOR4D_Cross(&u, &v, &n);
    // Все почти готово, осталось только нормировать
    // вектор нормали! Это основная оптимизация,
    // которую можно отложить. Чтобы оптимизировать
    // этот шаг, нужно предвычислить длину нормали

    // Вычисление длины нормали
    nl = VECTOR4D_Length_Fast(&n);

    // Напомним, как выглядит модель бесконечно
    // удаленного источника:  $I(d)dir = I_0dir * Cldir$ 
    // Модель диффузного освещения:
    //  $I_{totald} = R_{sdiffuse} * I_{diffuse} * (n * l)$ .
    // Чтобы избежать операций над числами с плавающей
    // запятой, умножим все на 128. Это делается не
    // потому, что операции над числами с плавающей
    // запятой выполняются медленнее, а потому, что
    // преобразование чисел одного типа в другой
    // требует много времени
    dp = VECTOR4D_Dot(&n, &lights[curr_light].dir);

    // Источник добавляется только при условии dp > 0
    if (dp > 0)
    {
        i = 128 * dp / nl;

        r_sum+= (lights[curr_light].c_diffuse.r *

```

```

        r_base * i) / (256*128);
    g_sum+= (lights[curr_light].c_diffuse.g *
        g_base * i) / (256*128);
    b_sum+= (lights[curr_light].c_diffuse.b *
        b_base * i) / (256*128);
} // if
} // if для бесконечно удаленного источника.
else if (lights[curr_light].attr & LIGHTV1_ATTR_POINT)
{
    // Вычисления для точечного источника:
    //      I0point * Clpoint
    //      I(d)point =
    //      kc + kl*d + kq*d^2
    //
    // где d - |p - s|. Эта модель почти идентична
    // модели бесконечно удаленного источника, однако
    // в ней интенсивность убывает с увеличением
    // расстояния от источника к освещаемому элементу
    // поверхности. Нужно вычислить нормаль к данному
    // элементу; напомним, что вершины задаются по
    // часовой стрелке: u=p0->p1, v=p0->p2, n=uxv
    VECTOR4D u, v, n, l;

    // Строим векторы u, v
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_1 ], &u);
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_2 ], &v);

    // Вычисление векторного произведения
    VECTOR4D_Cross(&u, &v, &n);
    // Все почти готово, осталось только нормировать
    // вектор нормали! Это основная оптимизация,
    // которую можно отложить. Чтобы оптимизировать
    // этот шаг, нужно вычислить длины всех нормалей

    // Вычисление длины нормали
    nl = VECTOR4D_Length_Fast(&n);

    // Строим вектор, направленный от элемента
    // поверхности к источнику
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &lights[curr_light].pos, &l);

    // Вычисляем расстояние и затухание
    dist = VECTOR4D_Length_Fast(&l);

    // Для диффузионной модели:
    // Itotald - Rsdiffuse*Idiffuse * (n*l).
    // Чтобы избежать операций над числами с плавающей
    // запятой, умножим все на 128. Это делается не
    // потому, что операции над числами с плавающей
    // запятой выполняются медленнее, а потому, что

```

```

// преобразование чисел одного типа в другой
// требует много времени
dp = VECTOR4D_Dot(&n, &l);

// Источник добавляется только при условии dp > 0
if (dp > 0)
{
    atten = (lights[curr_light].kc +
             lights[curr_light].kl*dist +
             lights[curr_light].kq*dist*dist);
    i = 128*dp / (nl * dist * atten);

    r_sum += (lights[curr_light].c_diffuse.r *
              r_base * i) / (256*128);
    g_sum += (lights[curr_light].c_diffuse.g *
              g_base * i) / (256*128);
    b_sum += (lights[curr_light].c_diffuse.b *
              b_base * i) / (256*128);
} // if
} // if point
else
if (lights[curr_light].attr &
    LIGHTV1_ATTR_SPOTLIGHT1)
{
    // Вычисления по упрощенной модели светового
    // пятна. Модель представляет собой точечный
    // источник с выделенным направлением:
    //      I0point * CLpoint
    //      I(d)point =
    //      kc + kl*d + kq*d^2
    //
    // где d - |p - s|. Эта модель почти идентична
    // модели бесконечно удаленного источника, но
    // интенсивность освещения убывает с ростом
    // расстояния от источника до освещаемой
    // поверхности. Нужно вычислить нормаль к данной
    // поверхности. Напомним, что вершины задаются
    // по ходу часовой стрелки, u=p0->p1, v=p0->p2,
    // n=uxv
    VECTOR4D u, v, n, l;

    // Построение векторов u, v
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                   &obj->vlist_trans[ vindex_1 ], &u);
    VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                   &obj->vlist_trans[ vindex_2 ], &v);

    // Вычисление векторного произведения (чтобы
    // определить нормаль)
    VECTOR4D_Cross(&v, &u, &n);
    // Все почти готово, осталось только нормировать
    // вектор нормали! Это основная оптимизация,
    // которую можно отложить. Чтобы оптимизировать

```

```

// Этот шаг, нужно предвычислить длину нормали
nl=VECTOR4D_Length_Fast(&n);

// Вычисление длины нормали
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
               &lights[curr_light].pos, &l);

// Вычисление расстояния и затухания
dist=VECTOR4D_Length_Fast(&l);

// Для диффузионной модели:
// Itotald = Rsdiffuse*Idiffuse * (n*l).
// Таким образом, все это нужно перемножить.
// Чтобы избежать операций над числами с
// плавающей запятой, умножим все на 128. Это
// делается не потому, что операции над числами
// с плавающей запятой выполняются медленнее, а
// потому, что преобразование чисел одного типа
// в другой требует много времени. Заметим, что
// вместо вектора, задающего направление к
// источнику, используется вектор направления от
// источника, а это нужно учитывать
dp=VECTOR4D_Dot(&n, &lights[curr_light].dir);

// Источник добавляется только при выполнении
// условия dp > 0
if (dp > 0)
{
    atten = (lights[curr_light].kc +
             lights[curr_light].kl*dist+
             lights[curr_light].kq*dist*dist);

    i=128*dp / (nl * atten );

    r_sum += (lights[curr_light].c_diffuse.r *
              r_base * i) / (256*128);
    g_sum += (lights[curr_light].c_diffuse.g *
              g_base * i) / (256*128);
    b_sum += (lights[curr_light].c_diffuse.b *
              b_base * i) / (256*128);
} // if
} // if spotlight1
else
if (lights[curr_light].attr &
    LIGHTV1_ATTR_SPOTLIGHT2)
// Упрощенная модель светового пятна, но более
// сложная, чем предыдущая
{
    // Еще раз выполняем вычисления для упрощенной
    // модели светового пятна:
    // IDspotlight
    // 
$$I(d)_{spotlight} = \frac{1}{kc + kl*d + kq*d^2} *$$

    //

```

```

// * Clspotlight * MAX( (l . s), 0)^pf,
// где d = |p - s|, а pf = показатель степени.
// Эта модель почти идентична модели точечного
// источника, но в числителе находится один
// дополнительный член, зависящий от угла
// между направлением на световой источник и
// осью светового конуса

// Необходимо вычислить нормаль к поверхности
// многоугольника. Напомним, что вершины
// перечисляются по ходу часовой стрелки,
// u=p0->p1, v=p0->p2, n=uxv
VECTOR4D u, v, n, d, s;

// Построение векторов u, v.
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
    &obj->vlist_trans[ vindex_1 ], &u);
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
    &obj->vlist_trans[ vindex_2 ], &v);

// Вычисление векторного произведения (чтобы
// изменить знак n, вычисляем v x u)
VECTOR4D_Cross(&v, &u, &n);
// Все почти готово, осталось только
// нормировать вектор нормали! Это основная
// оптимизация, которую можно отложить. Чтобы
// оптимизировать этот шаг, нужно
// предвычислить длину нормали.
nl = VECTOR4D_Length_Fast(&n);

// Для диффузионной модели
// Itotald = Rsdiffuse * Idiffuse * (n . l).
// Чтобы избежать операций над числами с
// плавающей запятой, умножим все на 128. Это
// делается не потому, что операции над
// числами с плавающей запятой выполняются
// медленнее, а потому, что преобразование
// чисел одного типа в другой требует много
// времени
dp = VECTOR4D_Dot(&n, &lights[curr_light].dir);

// Источник добавляется только при выполнении
// условия dp > 0

if (dp > 0)
{
    // Построение вектора, направленного от
    // источника к поверхности (он отличен от
    // вектора l, который задает главное
    // направление источника)
    VECTOR4D_Build( &lights[curr_light].pos,
        &obj->vlist_trans[ vindex_0 ], &s);

    // Вычисление длины вектора s (расстояние к

```

```

// источнику); это нужно для его нормировки
dist - VECTOR4D_Length_Fast(&s);

//Вычисление множителя, специфического для
// светового пятна (s*t)
float dpsi - VECTOR4D_Dot(&s,
    &lights[curr_light].dir)/dist;

//Продолжаем вычисление, только если эта
// величина положительна
if (dpsi > 0)
{
    // Вычисление затухания
    atten = (lights[curr_light].kc+
        lights[curr_light].kl*dist+
        lights[curr_light].kq*dist*dist);

    // Чтобы ускорить вычисления, значения pf
    // < 1.0 не рассматриваются; кроме того,
    // показатель степени должен быть целым
    // числом
    float dpsi_exp = dpsi;

    // Возведение в степень с положительным
    // целым показателем
    for (inte_index = 1;
        e_index < (int)lights[curr_light].pf;
        e_index++)
        dpsi_exp*=dpsi;

    // В переменной dpsi_exp хранится степень
    // (dpsi)^pf, которая равна (s*t)^pf
    i - 128*dp * dpsi_exp / (nl * atten );

    r_sum += (lights[curr_light].c_diffuse.r
        * r_base * i) / (256*128);
    g_sum += (lights[curr_light].c_diffuse.g
        * g_base * i) / (256*128);
    b_sum += (lights[curr_light].c_diffuse.b
        * b_base * i) / (256*128);
} // if
} // if
} // if spot light
} // for Light

//Убедимся, что значения цвета лежат в допустимом
// диапазоне
if (r_sum > 255) r_sum - 255;
if (g_sum > 255) g_sum - 255;
if (b_sum > 255) b_sum - 255;

// Запись цвета.
shaded_color - RGB16Bit(r_sum, g_sum, b_sum);
curr_poly->color = (int)((shaded_color << 16) |
    curr_poly->color);

```

```

} // if
else // Считаем, что это -
    // POLY4DV1_ATTR_SHADE_MODE_CONSTANT
{
    // Только для излучательного затенения. Копирование
    // базового цвета в верхние 16 битов (без изменений)
    curr_poly->color = (int)((curr_poly->color << 16) |
        curr_poly->color);
} // else

} // for poly

// Успешное выполнение.
return(1);

} // Light_OBJECT4DV1_World16

```

Функция, моделирующая освещение, представляет собой бесхитростную реализацию того, что было описано в начале данной главы. По сути, сначала в функции определяется формат пикселя, а затем извлекаются RGB-компоненты цвета многоугольника, после чего выполняется переход в 8-битовый режим. Далее идет цикл по источникам освещения. Если для данного многоугольника осуществляется постоянное затенение — переходим к следующему многоугольнику. В противном случае многоугольник освещается с учетом параметров, заданных для источника. В ходе выполнения алгоритма суммируются вклады всех источников, которые активны в данном списке источника — общих, точечных и бесконечно удаленных. В данной функции поддерживаются даже световые пятна, но в очень упрощенной модели. Можно заметить, что есть два набора параметров для световых пятен. Это сделано для того, чтобы обеспечить поддержку двух различных моделей световых пятен. Эти модели описаны ниже.

- **LIGHT\_ATTR\_SPOTLIGHT1.** Световое пятно этого типа похоже на точечный источник. Отличие состоит в том, что в данной модели используется вектор, задающий направление источника. Таким образом, параметры этого источника подобны параметрам точечного источника, однако для него также нужно задавать координаты вектора, указывающего направление. Интересно, что имитация эффекта затухания хорошо воспроизводит уменьшение интенсивности в световом конусе.
- **LIGHT\_ATTR\_SPOTLIGHT2.** Эта упрощенная модель в основном воспроизводит полную модель светового пятна, однако в ней отсутствуют явно заданные внутренний и внешний световые конусы. Если вы помните, ранее были описаны две модели светового пятна, в одной из которых использовался показатель степени, в которую возводится косинус угла между осью светового конуса и направлением на освещаемую точку. Этот показатель определяет, в какой степени интенсивность освещения сконцентрирована возле оси конуса. Таким образом эффективно контролируется радиус светового пятна. В данной модели задаются все параметры светового пятна, за исключением внутреннего и внешнего углов светового конуса. Кроме того, показатель степени  $r^d$  должен быть целым числом, большим 1.0. Возведение в степень выполняется путем последовательного умножения, поскольку вызов функции  $\exp()$  может оказаться слишком долгим.

После вычисления интенсивностей каждого источника, имеющегося в базе данных, их вклады суммируются, а затем полученные значения в формате RGB преобразуются в

формат 5.6.5 или в формат 5.5.5, в зависимости от формата пикселя. Это значение копируется в верхние 16 битов поля, в котором хранится цвет многоугольника, где оно может находиться без риска быть перезаписанным реальным цветом многоугольника. Все!

**ВНИМАНИЕ**

В процессе вычисления интенсивности **освещения** неоднократно возникает риск **переполнения** RGB-каналов. Поэтому необходимо выполнять **соответствующие** проверки, не допуская переполнения (т.е. интенсивность ни в одном из каналов не должна превысить 255 или **другого** максимально допустимого значения). Для этих проверок, конечно же, следует использовать встраиваемую функцию.

К сожалению, эта функция так проста, что ситуации, когда ее можно вызвать, крайне редки. При вызове функции необходимо указать освещаемый объект, задать параметры камеры (пока что камера не используется, однако она может понадобиться позже), указать используемые источники, а также их количество. Ниже представлен типичный вызов функции. Предполагается, что параметры источников занесены в глобальный массив с именем `lights` и что активны три источника.

// Выполняем освещение.

```
Light_OBJECT4DV1_World(&obj_marker, &cam, lights, 3);
```

В результате такого вызова объект будет **освещен**, а цвета освещенных многоугольников будут записаны в старшие 16 битов **соответствующих** переменных. После этого объект можно будет вставить в список визуализации с помощью функции `Insert_OBJECT4DV1_RENDERLIST4DV2()`.

Функция, которая работает почти так же, но не с самим объектом, а с элементами списка визуализации, имеет следующий прототип.

```
int Light_RENDERLIST4DV1_World16(  
    // Обрабатываемый список визуализации  
    RENDERLIST4DV1_PTR rend_list,  
    CAM4DV1_PTR cam,           // Положение камеры  
    LIGHTV1_PTR lights,        // Список источников (их  
                                // может быть несколько)  
    int max_lights);           // Максимальное количество  
                                // источников в списке
```

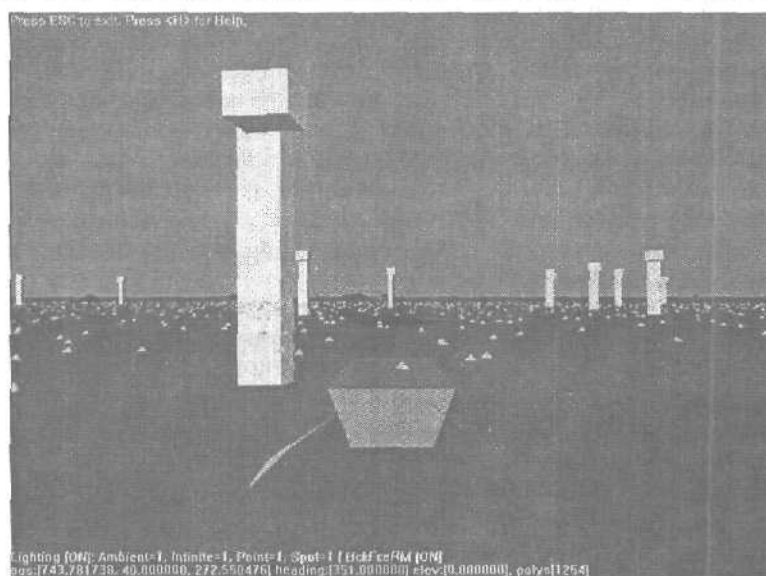
Эта функция используется так же, как и функция, моделирующая освещение на уровне объекта. Однако ее следует вызывать после удаления обратных поверхностей и тогда, когда список визуализации задан в глобальных координатах или в системе отсчета камеры.

Продemonстрируем, как работает функция, моделирующая **освещение**. На рис. 8.32 приведен снимок экрана, полученный в результате запуска программы `DEMO118_5.CPP[EXE]`. В этой программе создаются: источник общего света, точечный источник, бесконечно удаленный источник и световое пятно типа 1, которые размещаются в глобальной системе координат. В начале работы **демонстрационной** программы все эти источники становятся активными, и наблюдатель получает возможность увидеть их в процессе имитации. Кроме того, на танках установлены вертикальные стабилизаторы, для которых задаются излучаемые цвета (другими словами, постоянное затенение), поэтому они выглядят как источники света, которые не излучают. Список клавиш, с помощью которых производится управление работой демонстрационной программы, приведен в табл. 8.2.

Эта демонстрационная программа замечательно работает. Я люблю наблюдать, как по экрану передвигаются источники света, особенно точечные источники и световые пятна (они еще и цветные)! Желаю и вам получить удовольствие, а также полезный опыт, наблюдая за огоньками, изменяя их **цвет** и т.п. Удивительно, что данная демонстрационная программа, моделирующая **освещение** цветными источниками в реальном времени и в 16-битовом режиме, работает, причем работает неплохо!

**Таблица 8.2. Управляющие клавиши программы DEMO18 5.CPP|EXE**

Клавиша	Функция
<A>	Переключает источник общего света
<I>	Переключает бесконечно удаленный источник
<P>	Переключает точечный источник
<S>	Переключает световое пятно
<W>	Переключает режим отображения (проволочный каркас/заполненный)
<B>	Включает и выключает режим удаления обратных поверхностей
<L>	Включает и выключает игровой процессор освещения
Стрелка вправо	Поворачивает сцену вправо
Стрелка влево	Поворачивает сцену влево
Стрелка вверх	Приближает камеру к сцене
Стрелка вниз	Удаляет камеру
Пробел	Турборежим
<H>	Вызов команды меню Help
<Esc>	Выход из программы



*Рис. 8.32. Экранный снимок, полученный в процессе работы игрового процессора, моделирующего освещение*

## Плоское затенение в 8-битовом режиме

Реализация освещения в 8-битовом режиме приводит к определенным проблемам. Это вызвано несколькими причинами. Первая из них заключается в том, что для цвета выделено всего 8 битов! Однако основная проблема — та, что у нас в лю-

бом случае имеется только одна палитра. Независимо от того, каким образом освещается поверхность, в конечном итоге все будет выражено в 8-битовом режиме, причем при помощи исходной палитры (палитру мы не меняем, по крайней мере, в пределах одной сцены игры). С учетом сказанного, есть несколько подходов к написанию функции затенения, работающей в 8-битовом режиме. Например, можно предположить, что все источники излучают белый цвет, а также упростить модель источника, задавая для него только интенсивность, но не цвет, и т.д. Вспомните, как мы строили таблицы соответствия цветов (см. рис. 8.24-8.26). Определенно, лучше использовать вторую таблицу, в которой представлены различные цвета, и в каждой строке по 256 интенсивностей цвета. Таким образом, чтобы ускорить преобразование цветов, можно использовать следующую схему.

```
shaded_color = rgbilookup[color][shade];
```

Конечно же, и цвет, и его интенсивность нумеруются индексами, которые лежат в интервале 0..255. Это приводит к тому, что результирующий 8-битовый индекс дает нам цвет, который лучше всего соответствует реальной палитре цветов.

Однако я не хочу на данном этапе накладывать ограничения на систему освещения. Я очень надеюсь, что 16-битовая версия будет оставаться достаточно быстрой, поэтому пока что мы не собираемся оптимизировать ее, преобразуя к 8-битовому виду. Мы собираемся поддерживать ее, хотя и немного не в том виде, в котором вы ее видели. Заметим, что для разработки 8-битового модуля освещения недостаточно просто заменить несколько строк 16-битовой версии кода другими. План будет таким: о формате пикселей больше беспокоиться не нужно; мы просто будем работать с индексом, задающим цвет многоугольника. Основываясь на этом индексе, найдем с помощью палитры фактические RGB-значения цвета, а затем используем их (предварительно выразив в 8-битовом режиме) в вычислениях, связанных с моделированием освещения. После завершения этих вычислений полученные RGB-значения будут преобразованы в индексы, выраженные в кодировке 5.6.5, и использованы в таблице, созданной с помощью вызова функции `RGB_16_8_IndexedRGB_Table_Builder()`. Если вы помните, эта функция берет произвольное RGB-значение, закодированное в 16-битовом формате, а затем, используя это значение как индекс, выдает индекс, который соответствует наиболее близкому цвету из палитры. Схема этого процесса приведена на рис. 8.33.

У нас нет места для того, чтобы привести здесь всю функцию (к тому же она почти идентична той, что работает в 16-битовой версии). Ограничимся тем, что представим ее прототип.

```
int Light_OBJECT4DV1_World(OBJECT4DV1_PTR obj,
                           // Обрабатываемый объект
                           CAM4DV1_PTR cam, // Положение камеры
                           LIGHTV1_PTR Lights, // Список источников (их может
                                                // быть несколько)
                           int max_lights); // Количество источников в списке
```

#### НА ЗАМЕТКУ

Напомним, что для того, чтобы легче было различать 16- и 8-битовые версии функций, принято соглашение, согласно которому имя 16-битовой версии оканчивается на 16. Функции выглядят и работают одинаково, однако их имена немного различаются. Эти две функции, конечно же, можно было бы объединить в одну, но лучше, чтобы **это были** разные функции — тогда они имеют меньший размер и работают быстрее.

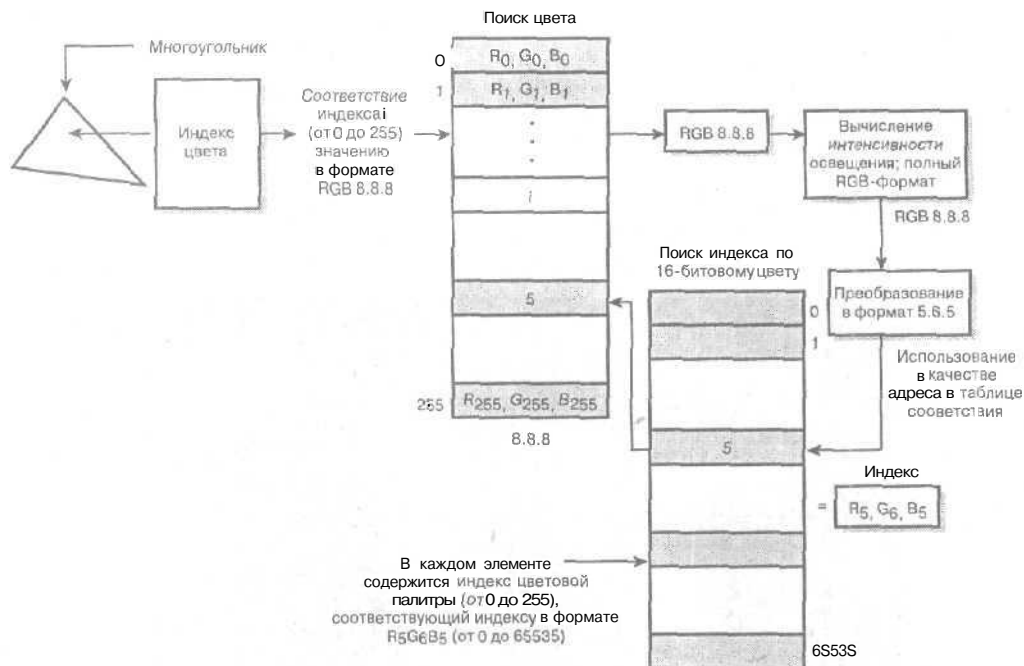


Рис. 8.33. Процесс освещения в 8-битовом формате

Приведем прототип 8-битовой функции, которая заносит объект в список визуализации. Еще раз отметим, что она работает так же, как и 16-битовая версия.

```
int Light_RENDERLIST4DV1_World(
    RENDERLIST4DV1_PTR rend_list, // Текущий список
                                // визуализации
    CAM4DV1_PTR cam,             // Положение камеры
    LIGHTV1_PTR lights,           // Список источников (их может быть
                                // несколько)
    int max_lights);              // Максимальное количество
                                // источников в списке
```

Чтобы понять, как работает 8-битовая система освещения, ознакомьтесь с демонстрационной программой DEMO118\_5\_8b.CPP|EXE. Работа этой программы не отличается от ее 16-битовой версии (в ней используются те же управляющие клавиши), за исключением того, что эта программа 8-битовая! Обратите внимание, что ухудшение качества изображения не столь значительное по сравнению с 16-битовой версией. На рис. 8.34 приведена копия экрана при работе этой демонстрационной программы.

Однако здесь следует учесть особенности палитры. В данной программе используется палитра, которая использовалась по умолчанию в книге Программирование игр для Windows, Советы профессионала. Эта палитра находится в файле PALDATA2.PAL. Чтобы просмотреть цвета палитры, можно загрузить файл PALDATA2.BMP. На первый взгляд палитра выглядит хорошо, однако это справедливо только для тех цветов, которые в ней содержатся. Если же попытаться подобрать оттенок отсутствующего в палитре цвета фуксии, то получим светло-коричневый или какой-то другой ужасный цвет, и качество изображения будет плохим.

Чтобы избежать этого, необходимо использовать палитру, в которой **мало** цветов, но много их оттенков. Например, в процессе разработки игры типа *Doom* можно было бы выбрать палитру с рядом серых, коричневых и других подобных **цветов**; при этом в палитре должно быть по 8-16 оттенков каждого из них. Если пойти таким путем, т.е. выбрать палитру с множеством всевозможных оттенков каждого цвета, в таблице соответствия не будет нелепых элементов.

Ну что же, для первого знакомства с моделью заполненных многоугольников и затенения было сказано достаточно. Теперь посмотрим, как можно улучшить модели плоского затенения и дополнить их на фазе растеризации другими моделями.

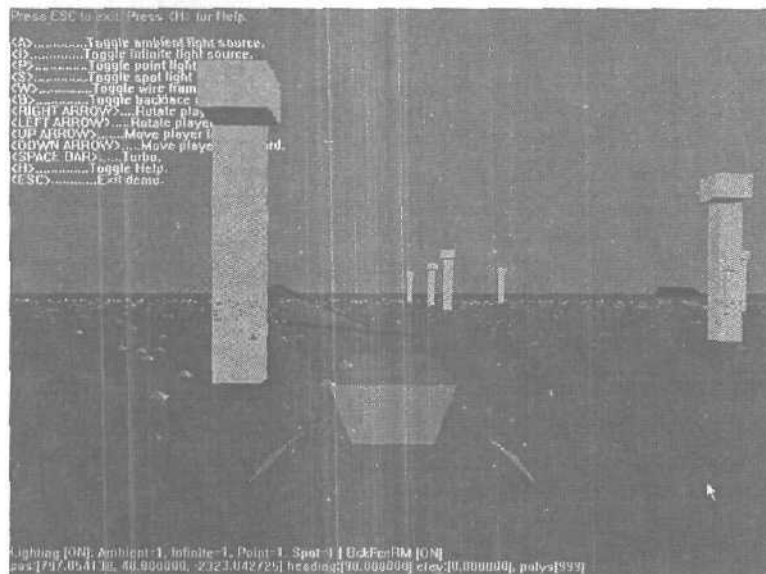


Рис. 8.34. Снимок экрана, полученный в ходе работы 8-битовой версии программы, моделирующей освещение

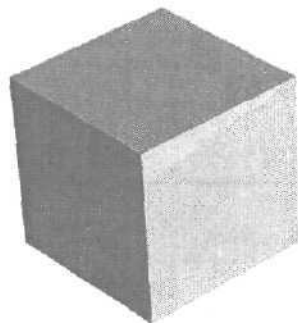
## Затенение по Гуро

Возможно, вы уже обратили внимание, что до сих пор в описываемых программах поддерживался только режим затенения, известный **под** названием *плоское затенение*. Это означает, что весь многоугольник выводится одним и тем же цветом; однако объекты при этом выглядят гранеными.

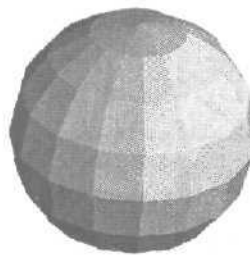
Конечно же, для объектов, которые **являются** гранеными, так и должно быть. Например, наша модель **освещения** здорово работает для куба, однако плохо — для сферы (рис. 8.35). Мы хотим сгладить границы цветов при переходе от одного элемента поверхности к другому, чтобы гладкие поверхности такими и **выглядели**. Обратите внимание на рис. 8.36. На нем изображена та же сфера, но на этот раз она затенена по Гуро. Посмотрите, какая она гладкая. Однако на этом же рисунке изображен куб, также затененный по Гуро. Он выглядит гладким, а это никуда не годится. Таким образом, для объектов с четко выраженными гранями затенение по Гуро применяться не должно.

Итак, как же работает затенение по Гуро? Как показано на рис. 8.37, сначала вычисляется цвет и интенсивность в каждой вершине каждого многоугольника. Для соприкасающихся многоугольников проводится процедура усреднения нормалей — они

усредняются в каждой вершине многоугольника, причем усреднение в вершине выполняется по всем многоугольникам, которым принадлежит эта вершина. После этого освещение моделируется как обычно, в результате чего каждой вершине присваивается свой цвет. Далее начинается самое интересное: многоугольник затеняется не одним и тем же цветом, а в соответствии с **интерполяционной** процедурой. Цвет в каждой вершине берется в качестве опорного, а во внутренней области многоугольника выполняется интерполяция по двум направлениям. Этот процесс проиллюстрирован на рис. 8.38. Несмотря на то, что определение цвета выполнялось не для каждого пикселя, **составляющего** многоугольник, все выглядит так, как если бы каждый пиксель обрабатывался отдельно! Здорово, правда?

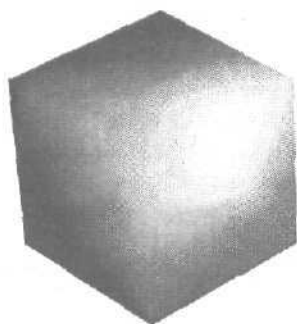


а. Куб с плоским затенением

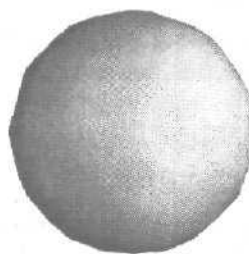


б. Сфера с плоским затенением

**Рис. 8.35. Объекты с плоским затенением**



а. Куб с затенением по Гуро



б. Сфера с затенением по Гуро

**Рис. 8.36. Объекты со сглаженным затенением**

Конечно же, такой способ обладает недостатками. Затенение выполняется на этапе растеризации каждого многоугольника. Таким образом, оно осуществляется в экранном пространстве, что приводит к искажению перспективы. К счастью, наши глаза не в состоянии рассмотреть этого (по крайней мере, не так хорошо, как текстуры), поэтому это не беда. Вторая проблема заключается в необходимости выполнения интерполяции, что может привести к значительному потреблению ресурсов на этапе растеризации. Однако интерполяция — это именно то, что нам нужно при построении текстурных карт, поэтому все это можно **выполнять** одновременно.

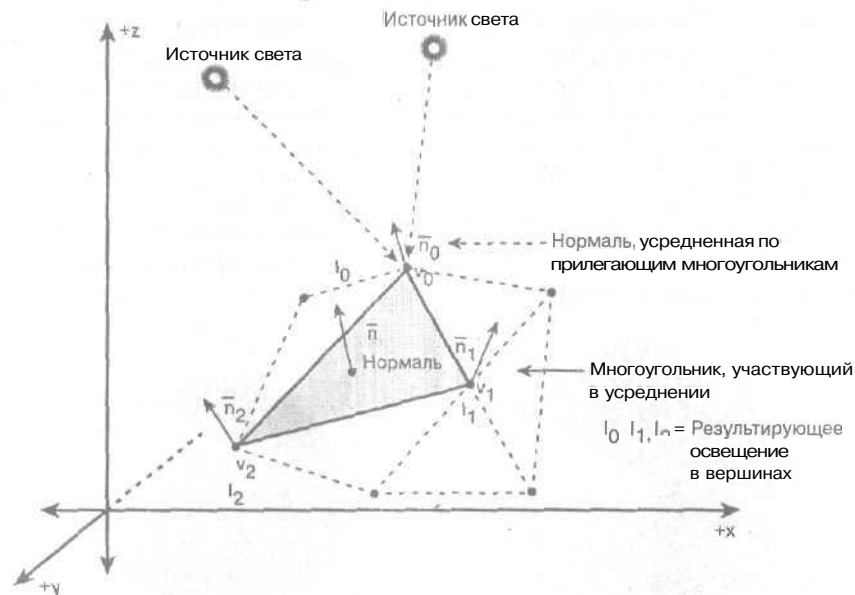


Рис. 8.37. Схема затенения по Гуро

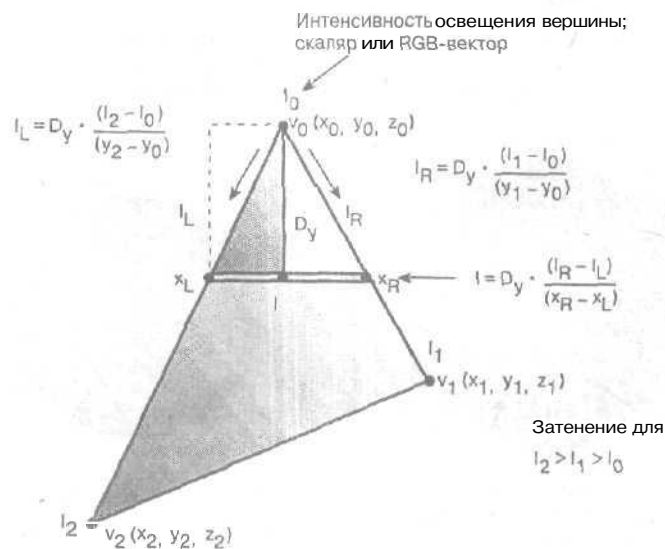


Рис. 8.38. Схема интерполяции при затенении по Гуро

Наконец, никто не говорил, что интенсивности **освещения** необходимо вычислять в каждой вершине. При желании соответствующие значения можно присваивать вручную. Другими словами, как моделировать статическое **освещение**, т.е. задавать его вручную или получать как результат расчетов — это ваш выбор. Перед тем как продолжить освоение материала, вкратце рассмотрим, по каким математическим формулам выполняется интерполирование интенсивности освещения многоугольника в процессе его затенения и растеризации. На рис. 8.38 изображен треугольник, который лежит в плоскости **рисун-**

ка, а также обозначены различные величины. Интерполирование **цветов** — это простая процедура. Сначала выполняется интерполяция для левой и правой сторон треугольника, в результате которой получаем значения величин  $I_L$  и  $I_R$  для каждой точки, принадлежащей этим сторонам. При этом используются только ординаты точек. Затем в ходе растеризации на основе **текущего** положения каждой **лежащей** на стороне точки  $(x, y)$  вычисляется интенсивность пикселей, находящихся в текущей строке развертки.

Надеюсь, этот алгоритм хорошо вам знаком — ведь он лежит в основе многих интерполяционных вычислений, предназначенных для цвета, текстуры и других объектов. В данной книге он будет интенсивно использоваться. В следующей главе, в которой речь пойдет об аффинных текстурах, этот алгоритм будет исследован более подробно. Поэтому если вы не совсем поняли его, не огорчайтесь — у вас еще будет такой шанс.

## Затенение по Фонгу

Затенение по **Гуро** приобрело статус фактического стандарта. Оно используется в 99% представленных на рынке графических ускорителей. Однако такое затенение не лишено **недостатков**, так как не принимает во внимание перспективу. Для учета этого и других подобных эффектов существует затенение по Фонгу. В нем затенение вычисляется для каждого отдельного пикселя. При выполнении затенения по Фонгу вместо интерполяции значений цвета в пространстве экрана в этом же пространстве интерполируются векторы нормали по набору внутренних точек многоугольника, а затем с помощью этих нормалей вычисляется **освещение** каждого пикселя. Этот процесс проиллюстрирован на рис. 8.39. Затенение по Фонгу выглядит несколько реалистичнее, чем затенение по Гуро. Оно значительно лучше передает зеркальные эффекты. При затенении по Гуро невозможно получить зеркальные области, поскольку в ходе линейной интерполяции **интенсивности** освещения, например, внутренней области многоугольника представляют собой промежуточные значения **освещенности** каждой вершины. А при затенении по Фонгу затенение выполняется для каждого пикселя, поэтому здесь возможны более резкие **переходы**.

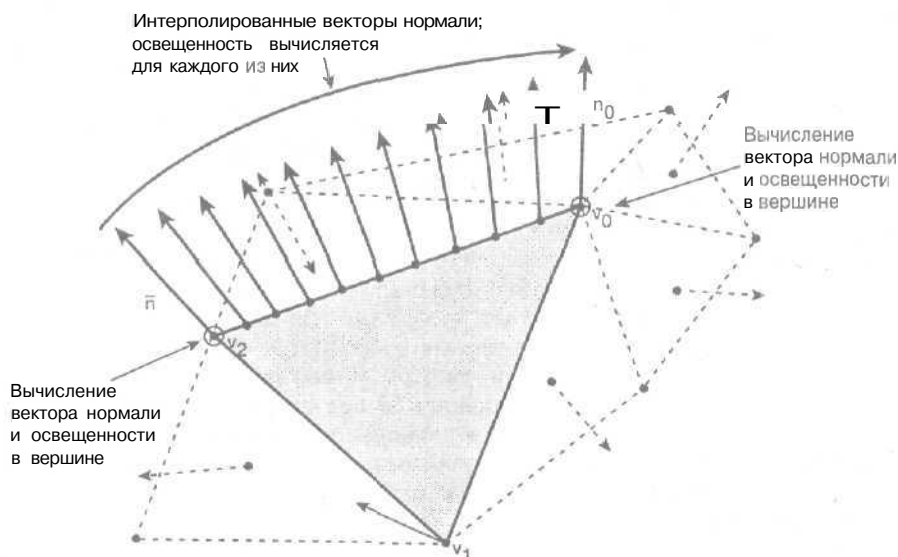


Рис. 8.39. Интерполирование нормалей при затенении по Фонгу

Конечно же, *здесь* есть и отрицательная сторона. Она заключается в том, что затенение по Фонгу вряд ли удастся выполнить на аппаратном уровне. Остается только программный уровень, так что мы не станем применять этот вид затенения во всех программах, которые встречаются в данной книге. Однако мы будем прибегать к некоторым трюкам, чтобы освещение было похоже на *то*, которое получается при затенении по Фонгу (это основная причина, по которой было описано данное затенение). Это будет тогда, когда мы достаточно далеко продвинемся в моделировании освещения.

## Сортировка по глубине и алгоритм художника

К этому времени вы получили достаточно подробные знания о трехмерных изображениях. Однако в обсуждении не был учтен один вопрос — видимость объекта и порядок визуализации. Когда речь идет о трехмерных изображениях, термин *видимость* (*visibility*) применяется довольно часто, однако его значение в основном сводится к тому, способны ли мы видеть данный многоугольник. В числе других факторов важную роль играет порядок вывода многоугольников на экран. Дело в том, что даже после отбраковки объектов, выходящих за поле зрения, и удаления обратных поверхностей следует сгенерировать общий список многоугольников и выводить их на экран именно в том порядке, в котором они идут в этом списке. Сама программа визуализации не имеет понятия о том, в каком порядке ей выводить многоугольники! Нет сомнений в том, что этот вопрос заслуживает внимания, и мы будем на протяжении нескольких глав обсуждать сложные алгоритмы, помогающие определить правильный порядок вывода. По ходу мы сможем понять методы, позволяющие отбросить еще больше геометрических элементов. Чтобы убедиться в том, что порядок вывода является важным, еще раз запустите рассмотренные до сих пор *демонстрационные* программы. Буквально через несколько секунд вы увидите, что объекты, которые должны находиться позади других, на самом деле изображены впереди них, и наоборот. Мягко выражаясь, это *сухая* неразбериха. В режиме, когда объекты выводятся в виде каркасов, все выглядит *нормально*, но стоит нам включить затенение — и картина резко меняется в худшую сторону.

### СОВЕТ

Напомним, что не следует *выводить* объекты, которые не видны!

Теперь мы собираемся реализовать простейший из алгоритмов, применяемых в ходе визуализации, который известен как алгоритм художника, и доведем программу до рабочего состояния (по крайней мере, насколько позволит наш уровень знаний). После этого можно будет продолжать работать с текстурами и освещением, не беспокоясь о неприятностях, связанных с неправильным порядком вывода объектов на экран.

Поговорим об алгоритме художника. По сути, он моделирует процесс нанесения красок на холст: сначала рисуется фон, а затем поверх него — передний план (рис. 8.40). Таким образом, объекты, которые находятся ближе к наблюдателю, автоматически закрывают задний план.

Поэтому данный *алгоритм* — это все, что нужно для надлежащего вывода многоугольников на экран. Нам известны значения координат *z*, соответствующих каждому многоугольнику из списка визуализации, поэтому можно просто отсортировать многоугольники в порядке уменьшения расстояния от них до наблюдателя (см. рис. 8.28). В большинстве случаев такой метод работает, однако иногда случаются досадные аномалии (рис. 8.41). Иногда неправильно визуализируются длинные, перекрывающиеся и другие аналогичные многоугольники. Сейчас мы не станем рассматривать этот вопрос, потому что наша первоочередная *задача* — улучшить изображение на дисплее. Кроме *того*, пока размеры многоугольников остаются незначительными, и вогнутые многогранники (такие, как на рис. 8.42) отсутствуют, алгоритм художника работает, как положено. Этот алгоритм (или его разновидности) используется во многих трехмерных играх.

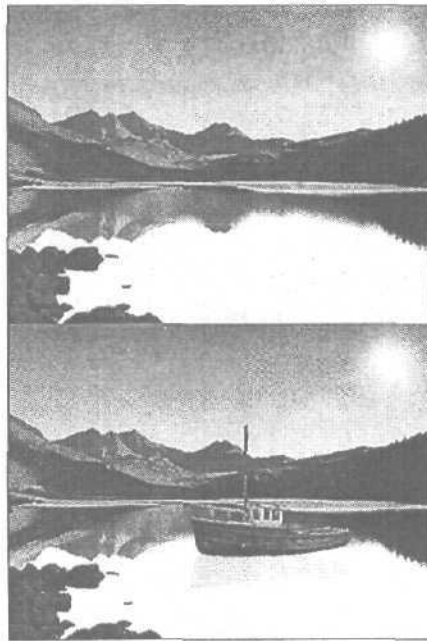


Рис. 8.40. Процесс создания холста

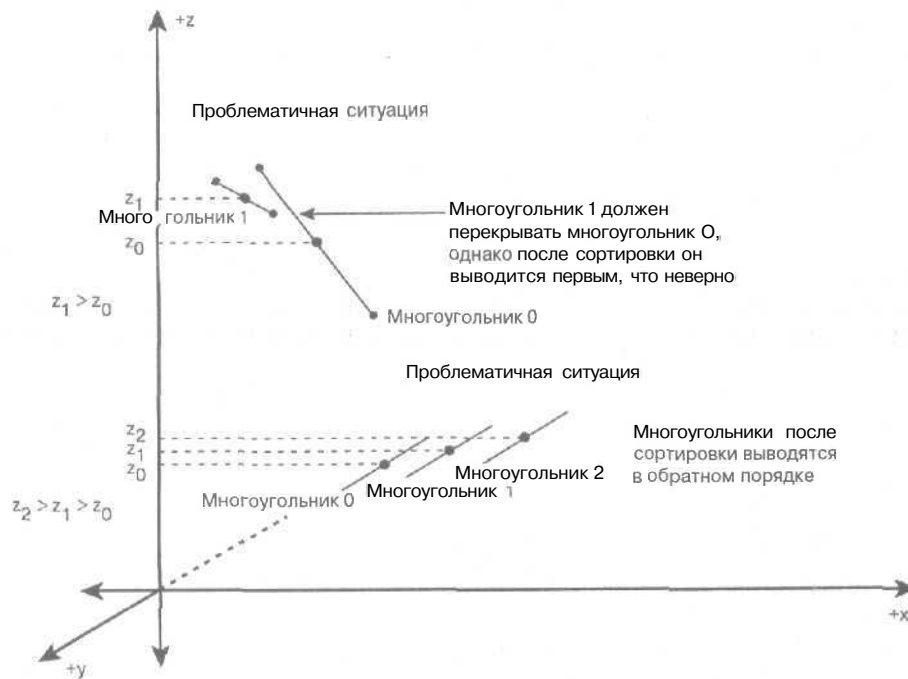
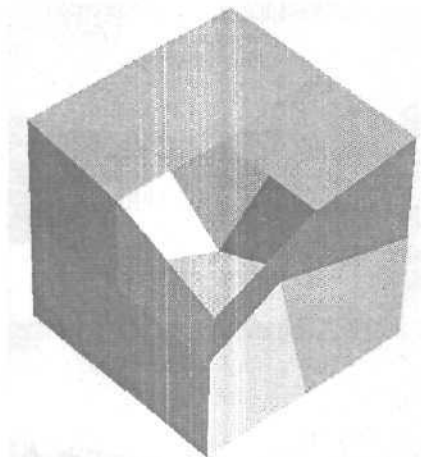


Рис. 8.41. Особые случаи, в которых  $z$ -сортировке не работает



*Рис. 8.42. Наличие вогнутых многогранников служит причиной многочисленных аномалий визуализации, осуществляемой по алгоритму художника*

НА ЗАМЕТКУ

Фактически алгоритм z-буферизации, который рассматривается далее в этой книге, представляет собой алгоритм художника, реализованный на уровне пикселей.

А теперь начинается самое интересное: подумаем, как реализовать рассматриваемый алгоритм для нашей системы и имеющихся в наличии структур данных. Это можно сделать с помощью двух методов. В первом производится сортировка многоугольников в списке визуализации, в результате чего каждый из них помещается на свое место. Во втором методе в списке визуализации создается вторичный массив индексов или указателей, которые затем сортируются. Первый метод может показаться более простым, однако перемещать сами данные слишком неэффективно.

Кроме того, в списке визуализации могут оказаться неактивные многоугольники, которые не нужно сортировать. Таким образом, мы займемся созданием вторичного списка указателей, каждый из которых указывает на многоугольник, содержащийся в списке визуализации. После этого указатели будут сортироваться, а визуализация будет производиться не на основании самого списка визуализации, а на основании вторичного списка. Описанный способ проиллюстрирован на рис. 8.43.

Если теперь взглянуть на структуру данных исходного списка визуализации, можно заметить, что она реализована именно таким образом.

```
typedef struct RENDERLIST4DV1_TYP
{
    int state; // Состояние списка визуализации
    int attr; // Атрибуты списка визуализации

    // Список визуализации - это массив указателей, каждый
    // из которых указывает на самодостаточный
    // визуализируемый элемент поверхности многоугольника
    // POLYF4DV1
    POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

    // Кроме того, чтобы сэкономить ресурсы, затрачиваемые
```

```

// на размещение и удаление каркасов каждого
// многоугольника, поверхности многоугольников будут
// храниться в массиве
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];
int num_polys; // Количество многоугольников в списке
// визуализации
} RENDERLIST4DV1, *RENDERLIST4DV1_PT;

```

Список визуализации

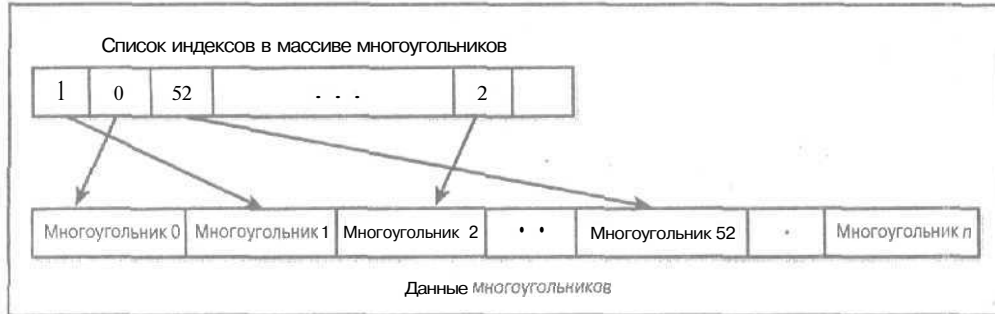


Рис. 8.43. Визуализация на основе массива индексов, а не на основе данных

Обратите внимание на код, выделенный полужирным шрифтом. Это и есть **дополнительный массив**, так что мы готовы к **продолжению**. Теперь нам нужен алгоритм сортировки. Воспользуемся **встроенным** алгоритмом быстрой сортировки, прототип которого приведен ниже.

```

void qsort(void *base, // Начало сортируемого массива
size_t num, // Размер массива (число элементов)
size_t width, // Размер элемента в байтах
int (cdecl *compare) // Функция сравнения элементов,
(const void *elem1, // указатели на которые переданы
const void *elem2)); // ей в качестве параметров

```

**СОВЕТ**

Напомним, что время, которое затрачивается на работу алгоритма быстрой сортировки, увеличивается с ростом числа элементов в сортируемом массиве по закону  $O(n \log n)$ . Данная функция работает по алгоритму рекурсивной сортировки, в котором весь список разбивается на части. Сначала сортируется каждая часть, после чего происходит повторная сборка.

Несмотря на видимую легкость использования данной функции, работа с указателями может все осложнить. Если подробнее рассмотреть прототип функции, в которой выполняется сравнение, мы увидим, что в нее передаются указатели на сравниваемые элементы. Однако в нашем случае эти элементы сами являются указателями! Это означает, что в функции сравнения нужно разыменовать указатели типа `const void *`, как если бы они были указателями на указатели на поверхности многоугольников. Это нужно сделать правильно, поскольку ситуация немного запутанная.

Кроме того, сама функция сравнения заслуживает того, чтобы о ней было сказано несколько слов. Известно, что мы собираемся сравнивать координаты  $z$  различных многоугольников, однако какое именно значение следует принять в качестве этой координаты? Можно воспользоваться координатой одной из трех вершин треугольника — **максималь-**

ной или минимальной, а можно усреднить координаты всех трех вершин. Мы реализуем поддержку всех трех перечисленных вариантов, выбор которых будет осуществляться с помощью флага. Это удобно, поскольку в разных эпизодах игры лучшие результаты может давать тот или иной способ. Таким образом, в алгоритме сортировки будет использован оператор case. Ниже приведен код этого алгоритма.

```
void Sort_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
    int sort_method)
{
    // В этой функции список визуализации сортируется на
    // основе координат z многоугольников; конкретный метод
    // сортировки задается путем выбора управляющего флага
    switch(sort_method)
    {
        case SORT_POLYLIST_AVGZ: // Сортировка по
                                // усредненному значению
        {
            qsort((void *)rend_list->poly_ptrs,
                rend_list->num_polys,
                sizeof(POLYF4DV1_PTR),
                Compare_AvgZ_POLYF4DV1);
        } break;
        case SORT_POLYLIST_NEARZ: // Сортировка по
                                // координатам z самых
                                // близких вершин
        {
            qsort((void *)rend_list->poly_ptrs,
                rend_list->num_polys,
                sizeof(POLYF4DV1_PTR),
                Compare_NearZ_POLYF4DV1);
        } break;
        case SORT_POLYLIST_FARZ: // Сортировка по
                                // координатам z самых
                                // удаленных вершин.
        {
            qsort((void *)rend_list->poly_ptrs,
                rend_list->num_polys,
                sizeof(POLYF4DV1_PTR),
                Compare_FarZ_POLYF4DV1);
        } break;

        default: break;
    } // switch
} // Sort_RENDERLIST4DV1
```

Я уверен, что вы ожидали гораздо более сложного и большого фрагмента кода! Приведем теперь флаги сортировки,

```
#define SORT_POLYLIST_AVGZ 0 // Сортировка по усредненному
                             // значению
#define SORT_POLYLIST_NEARZ 1 // Сортировка по координатам z
                              // самых близких вершин
```

```
ftdefine SORT_POLYLIST_FARZ 2 // Сортировка по координатам z
// самых удаленных вершин
```

Наконец, приведем саму функцию сортировки (в данном случае сравниваются усредненные координаты *z* вершин).

```
int Compare_AvgZ_POLYF4DV1(const void *arg1,
                           const void *arg2)
{
    // В данной функции усредняются координаты z вершин
    // каждого многоугольника, а затем эти усредненные
    // значения используются в алгоритме сортировки
    // многоугольников по глубине
    float z1, z2;
    POLYF4DV1_PTR poly_1, poly_2;
    // Разыменование указателей на многоугольники
    poly_1 = ((POLYF4DV1_PTR *)arg1);
    poly_2 = ((POLYF4DV1_PTR *)arg2);

    // Усреднение координат z по всем вершинам
    // многоугольника 1
    z1 = (float)0.33333*(poly_1->tvlist[0].z +
                       poly_1->tvlist[1].z +
                       poly_1->tvlist[2].z);

    // То же самое для многоугольника 2
    z2 = (float)0.33333*(poly_2->tvlist[0].z +
                       poly_2->tvlist[1].z +
                       poly_2->tvlist[2].z);
    // Сравниваем z1 и z2; многоугольники сортируются в
    // порядке убывания z

    if (z1 > z2)
        return(-1);
    else
        if (z1 < z2)
            return(1);
        else
            return (0);
} // Compare_AvgZ_POLYF4DV1
```

Остался один вопрос: в каком месте разместить вызов функции сортировки? Это очень важно, ведь сортировка должна производиться в системе отсчета камеры, когда определится положение всех многоугольников перед их проецированием. Таким образом, функцию сортировки следует вызвать после перехода из глобальной системы координат в систему координат, связанную с камерой, но перед тем, как будет выполнено проецирование. Ниже приведен один из возможных вариантов.

```
// Удаление обратных поверхностей
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);

// Моделирование освещения
Light_RENDERLIST4DV1_World16(&rend_list, &cam, lights, 3);

// Переход из глобальной системы координат в систему
// отсчета камеры
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);
```

```
// Сортировка списка многоугольников
Sort_RENDERLIST4DV1(&rend_list, SORT_POLYLIST_AVGZ);
```

```
// Преобразование перспективы
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);
```

```
// Переход в систему отсчета экрана
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);
```

В качестве примера действующего кода рассмотрим программу DEMOII8\_6.CPP|EXE (ее 8-битовая версия хранится в файле DEMOII8\_6\_8b.CPP|EXE). Снимок экрана, полученный при работе этой программы, приведен на рис. 8.44. Из этого рисунка видно, что теперь многоугольники отсортированы надлежащим образом, и все выглядит, как следует (по крайней мере, в основном). Однако, передвигаясь по игровому пространству (управляющие клавиши остаются такими же, как в предыдущей демонстрационной программе), можно убедиться, что алгоритм художника иногда дает сбой. Можно также поэкспериментировать, меняя алгоритм сортировки (т.е. сортируя многоугольники не по усредненной координате z, а по координате ближайшей или самой удаленной вершины).

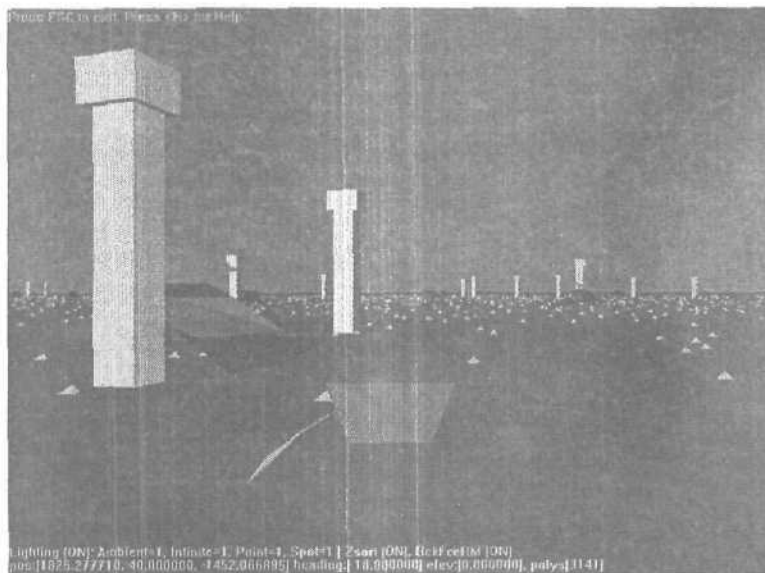


Рис. 8.44. Экранный снимок программы, в которой выполняется z-сортировка

#### СОВЕТ

Многоугольники в списке визуализации сортируются самым простым путем, но это не значит, что это самый быстрый путь упорядочения, так как эффективность сортировки может зависеть от сортируемых наборов данных. Можно предложить способ, в котором объекты в целом сортируются на основе окружающих их сфер. При этом объекты помещаются в список визуализации в порядке, полученном при сортировке. Другими словами, если объекты по большей части выпуклые и не пересекаются, их сортировку можно осуществлять на основе координат их центров. При этом правильный порядок визуализации получается за значительно меньшее время. Конечно, этот метод работает только в быстрых играх, сюжет которых развивается на большом пространстве и им подобных; в таких играх неправильное расположение нескольких многоугольников не приведет к непоправимым последствиям.

## Работа с новыми форматами моделей

До сих пор **вопрос**, о котором пойдет речь, не играл значительной роли, но теперь настало время реализовать **поддержку** более совершенных форматов, в которых формулируются модели. Заметим, что одно дело создавать ручную **кубы** и совсем другое — создавать **таким же** способом механических воинов, поэтому никто этим заниматься не **собирается!** Таким образом, в дополнение к уже использовавшимся нами форматам **.PLG** и **.PLX**, нужно организовать поддержку других форматов. Формат **PLG/PLX** замечательный. Его легко читать, в нем удобно манипулировать данными и анализировать их. Однако на рынке отсутствуют программные продукты, **поддерживающие** этот формат; кроме того, он обладает значительными ограничениями. Поэтому мы разработаем **еще** пару синтаксических анализаторов, пригодных для работы с другими имеющимися в наличии форматами файлов.

Однако для этого нужно решить один вопрос. Уже не в первый раз я затрудняюсь при выборе между эффективностью и наглядностью, и обычно побеждает наглядность. В **данном** случае важно приобрести полезные знания, а скучное написание бинарного синтаксического анализатора не наглядно; лично я предпочитаю работать с форматами, в **которых** данные представлены в текстовом виде. Такие данные можно просматривать и изменять вручную. Позже всегда можно будет самостоятельно написать бинарный синтаксический **анализатор**, а сейчас мы разработаем пару новых загрузчиков: один для файлов в формате **.ASC** (3D Studio Max ASCII-формат), а другой — в формате **.COB** (Caligari trueSpace, версия 4.0+). На прилагаемом компакт-диске можно найти статьи, посвященные этим форматам.

У обоих упомянутых форматов есть свои преимущества и недостатки. Однако оба они обладают достаточной гибкостью, необходимой для создания трехмерных моделей с помощью **реальных** инструментов (или использовать предварительно разработанные модели), которые впоследствии экспортируются в эти форматы. Даже если у вас нет программных продуктов 3D Max или trueSpace, наверняка можно найти конвертеры, преобразующие в данные форматы файлы, созданные **имеющимся** в наличии инструментарием. Перед тем, как продемонстрировать сами загрузчики файлов, ознакомимся с классом **C++**, представляющим собой синтаксический анализатор, а также написанной мною вспомогательной функцией, облегчающей анализ текстовых файлов.

### Класс-анализатор

Написание загрузчика файлов — **это**, в основном, тренировка в умении производить ввод-вывод файла независимо от того, с каким форматом должен будет работать этот **загрузчик** — ASCII или бинарным. Если потратить усилия на разработку набора функций, позволяющих загружать файлы, просматривать их содержимое и выполнять все необходимые операции, ваши усилия окупятся с лихвой. Поскольку мы будем работать с файлами в формате ASCII, **анализатор**, который мы собираемся создать, будет обладать такими возможностями:

- открывать и закрывать файл;
- считывать строку текста;
- извлекать символы из заданной строки;
- получать лексемы, разделенные пробельными символами;
- игнорировать все, что расположено после символов комментария;
- определять номер строки и выполнять другие подобные операции;
- выполнять сопоставление с образцом.

На рис. 8.45 изображена блок-схема разрабатываемого загрузчика. Как видно из этой схемы, загрузчик, **обладающий** перечисленными выше **возможностями**, будет нам весьма полезен. Обсудим последнюю из них, наиболее интересную.

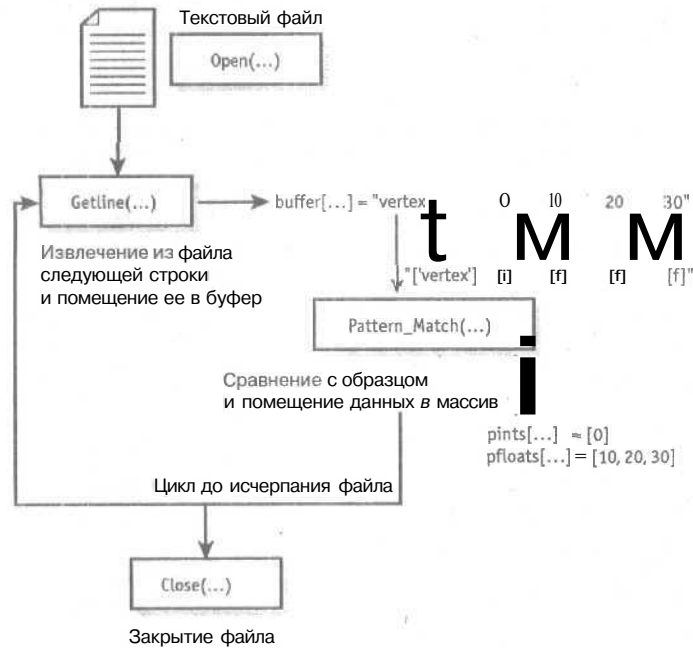


Рис. 8.45. Блок-схема загрузчика

Нам нужна система сопоставления с образцом, **позволяющая** находить в указанной строке обычные простые выражения. Пусть, например, в текстовом буфере содержится следующая строка.

```
static char *buffer="Polys: 10Vertices: 20";
```

Легко понять, что после надписи Polys: следует количество многоугольников (целое число), а после надписи Vertices: — количество вершин (целое число). Однако ни одна из функций `scanf()` не в состоянии проанализировать данную строку при наличии запятых или каких-либо других посторонних символов. Следовательно, нам нужен вспомогательный код, позволяющий производить сравнение с образцом. Эта дополнительная функциональная возможность реализована в моем классе анализатора. В результате можно написать команду, подобную ниже приведенной.

```
Pattern_Match(buffer, "[Polys:] [i] [Vertices:] [i]");
```

В результате функция, осуществляющая сопоставление с образцом, производит поиск строки "Polys:", считывает ее, затем — стоящее после этой строки целое **число**, далее она производит поиск строки "Vertices:", считывает ее, и наконец, — расположенное после нее целое число. После того как будет **приведен** код **класса-анализатора**, я вкратце ознакомлю вас с языком, позволяющим выполнять сравнение с образцом.

Нам также понадобятся вспомогательные функции, способные извлекать символы, производить их замену и т.п. По сути, нам понадобится более обширный набор операций со **строками**, чем тот, который предусмотрен в языках C и C++. Конечно же, есть множество готовых классов, однако если вы хотите научиться пользоваться процессорами

и классами, лучше научиться **создавать их** своими руками. **Ведь** вы читаете эту книгу именно для того, чтобы узнать, как все устроено!

Приведем определение класса-анализатора.

```
// Класс-анализатор
class CPARSERV1
{
public:
    // Конструктор
    CPARSERV1();
    // Деструктор
    ~CPARSERV1();
    // Сброс
    int Reset();
    // Открытие файла
    int Qopen(char *filename);
    // Закрытие файла
    int Close();
    // Доступ к строке
    char *Getline(int mode);
    // Задаем комментарий
    int SetComment(char *string);
    // Поиск строки по образцу
    int Pattern_Match(char *string, char *pattern, ...);

    // ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ
public:
    FILE *fstream;                // Указатель на файл
    char buffer[PARSER_BUFFER_SIZE]; // Буфер строки
    int length;                    // Длина текущей строки
    int num_lines;                 // Количество
                                  // обработанных строк
    char comment[PARSER_MAX_COMMENT]; // Одна строка
                                  // комментария

    // Хранилище совпавших образцов; с ним удобнее работать,
    // чем с аргументами функции. После выхода из функции
    // Pattern_Match () в этом хранилище будет находиться
    // все, что совпало с заданным образцом

    // Строки
    char pstrings[PATTERN_MAX_ARGS][PATTERN_BUFFER_SIZE];
    int num_pstrings;

    // Числа с плавающей точкой
    float pfloats[PATTERN_MAX_ARGS];
    int num_pfloats;

    // Целые числа
    int pints[PATTERN_MAX_ARGS];
    int num_pints;
}; // CLASS CPARSERV1
typedef CPARSERV1 *CPARSERV1_PTR;
```

Как я уже говорил, это очень простой класс, написанный на языке C++. Ненавижу пояснять программы, написанные на этом языке. Можно смотреть на код и не иметь представления, что он означает, поэтому я предпочитаю **пользоваться** языком C++ как можно реже. А теперь отвлечемся на минуту и ознакомимся с методами приведенного класса, а к нему самому мы **еще вернемся...**

Как видим, система предельно проста. Класс обладает функциями, которые перечислены ниже.

- `int Open(char *filename)` — функция открывает переданный ей файл. В случае успешного выполнения возвращается значение 1, а в случае сбоя — значение 0.
- `int Reset()` — обнуляет все переменные **класса-анализатора**, закрывает все файлы и **возвращает** класс в исходное состояние.
- `int Close()` — вызов функции `Reset()`.
- `char *Getline(int mode)` — если файл открыт для анализа, эта функция **осуществляет** доступ к его очередной строке. Если таковая существует, то функция возвращает эту строку; в противном случае она возвращает значение NULL. Вызов можно производить в следующих режимах.

```
#define PARSER_STRIP_EMPTY_LINES 1    // Удалить все пустые
// строки
#define PARSER_LEAVE_EMPTY_LINES 2    // Оставлять пустые
// строки
#define PARSER_STRIP_WS_ENDS 4        // Удалять пробел в
// конце строки
#define PARSER_LEAVE_WS_ENDS 8        // Оставлять пробел в
// конце строки
#define PARSER_STRIP_COMMENTS 16     // Удалять комментарии
#define PARSER_LEAVE_COMMENTS 32     // Оставлять комментарии
```

После завершения работы функции `Getline()` текущая строка заносится в переменной `buffer`, а сама функция возвращает указатель на эту строку.

- `int SetComment(char *string)` — эта функция позволяет задавать последовательность символов, с которой будут начинаться комментарии. Можно использовать произвольную последовательность, длина которой не превышает 16 символов, и в которой отсутствуют пробелы. В **большинстве** случаев хорошо подходят строки `#`, `;`, и `//`. По умолчанию применяется символ `#`.
- `int Pattern_Match(char *string, char *pattern,...)` — это самая сложная функция из всех. Она позволяет задавать образец, поиск которого должен производиться в данной строке. Заметим, что хотя функция `Pattern_Match()` является методом **класса-анализатора**, она позволяет указывать любую строку, в которой должно быть произведено сравнение с образцом. Элементы языка, с помощью которого выполняется такое сравнение, **приведены** в табл. 8.3.

**Таблица 8.3. Элементы языка, с помощью которого выполняется сравнение с образцом.**

Образец	Значение
[i]	Соответствует целое число
[f]	Соответствует число с плавающей точкой

Образец	Значение
[s=d]	Соответствует любая строка, длина которой составляет ровно d символов
[s<d]	Соответствует любая строка, длина которой меньше d символов
[s>d]	Соответствует любая строка, длина которой больше d символов
[^ssss...s <sup>1</sup> ]	Соответствует строка, совпадающая с той, что задана в одинарных кавычках

Например, если нужно найти строку, построенную по схеме  
 "Vertex 334.56 23.67 10.90"

то следует использовать образец "[Vertex] [i] [f] [f] [f]". Куда же будут помещены строки, которые подошли под заданный образец? Обратите внимание на выделенные полужирным шрифтом переменные и массивы, заданные в нижней части класса-анализатора. Например, если попытаться осуществить поиск чисел с плавающей запятой, то сами эти числа и их количество будут занесены в переменные

```
int pfloats[PATTERN_MAX_ARGS];
int num_pfloats;
```

Таким образом, в приведенном выше примере в результате сравнения с образцом будут выполнены следующие присвоения.

```
num_pfloats=3;
pfloats[]={34.56, 23.67, 10.90};
```

#### НА ЗАМЕТКУ

Как видите, функция `Pattern_Match()` заканчивается списком параметров переменной длины. Я собирался экспортировать найденные переменные в передаваемые переменные, однако не реализовал эту идею до конца, поскольку мне показалось, что вывод в массив работает лучше. Однако если впоследствии вам понравится эта идея, вы можете завершить ее реализацию.

Отметим несколько ограничений, присущих программе, которая выполняет сравнение с заданным образцом. Она работает, только если искомые последовательности символов отделены пробельными символами. Поэтому для строки

"Vertex 3: 20.9, 90.8, 100.3"

функция работать не будет! Помочь в этой ситуации могут вспомогательные функции. Рассмотрим их подробнее.

### Вспомогательные функции для синтаксического анализа

Ниже приведена небольшая библиотека, в которую входят вспомогательные функции, выполняющие различные операции со строками. Эту библиотеку также можно найти в файле `T3DLIB6.CPP`[Н]. Функции разработаны таким образом, что их можно использовать либо отдельно, либо совместно с классом-анализатором. Все зависит от вашего желания. Ниже перечислены все эти функции.

```
char buffer[64]; // Используется для вывода
```

Сначала рассмотрим функцию, удаляющую символы.

```
// Удаление символов из строки
int StripChars(char *string_in, char *string_out,
               char *strip_chars, int case_on=1);
```

В функцию `StripChars()` передается строка, в которой нужно удалить символы, а также сами удаляемые символы вместе с флагом, с помощью которого задается восприимчивость к регистру. Ниже приведен пример вызова функции, позволяющего удалить из строки все двоеточия и запятые.

```
StripChars("Vertex 3: 20.9, 90.8, 100.3", buffer, ":", ", 1);
```

В результате в буфер будет помещена строка "Vertex 3 20.9 90.8 100.3".

А теперь рассмотрим функцию, которая не удаляет символы, а заменяет их.

```
// Замена символов в строке
int ReplaceChars(char *string_in, char *string_out,
                char *replace_chars, char rep_char,
                int case_on=1);
```

Иногда удаление символов — это не совсем то, что нужно; бывает так, что их нужно просто заменить пробелом или другим символом. Для этого предназначена функция `ReplaceChars()`. Эта функция похожа на функцию `StripChars()`, но по сравнению с ней функция `ReplaceChars()` обладает одним дополнительным параметром типа `char`. С помощью этого параметра в функцию передаются символы, которые помешаются вместо заменяемых. В качестве примера рассмотрим следующую строку.

```
"Vertex #3: 20.9,90.8,100.3"
```

Если мы обработаем ее с помощью функции `StripChars()`, в качестве параметров которой будут заданы символы `#` и `:`, получим следующий результат.

```
"Vertex 320.990.8100.3"
```

Однако здесь возникает проблема, связанная с тем, что вместо запятых следовало бы вставить пробелы, а мы этого не сделали, и часть строки слилась. А вот с помощью функции `ReplaceChars()` можно добиться желаемого результата.

```
ReplaceChars("Vertex #3: 20.9,90.8,100.3",
            buffer, "#:", " ", 1);
```

В итоге в буфер будет помещена строка

```
"Vertex 3 20.9 90.8 100.3"
```

т.е. именно то, что нам нужно. Затем можно вызывать функцию `Pattern_Match(buffer, "[i][f][f][f]")`, осуществляющую поиск последовательности из одного целого числа и трех чисел с плавающей запятой.

Далее, у нас есть несколько функций для удаления лишних пробелов.

```
// Удаляет все пробелы, расположенные слева в строке
char *StringLtrim(char *string);
```

```
// Удаляет все пробелы, расположенные справа в строке
char *StringRtrim(char *string);
```

Следующие две функции определяют, является ли данная строка числом с плавающей запятой или целым числом. В случае положительного результата обе эти функции возвращают это значение. Эти функции не пытаются внести какие-либо исправления, поэтому они являются функциями, производящими сравнение с образцом. Если анализируемая строка не является числом с плавающей запятой или целым числом, то функции возвращают значения `FLT_MIN` и `INT_MIN`, соответственно. Эти значения определены в файле `limits.h` как ошибки.

```
// Функция, преобразующая строку в число с плавающей точкой
float IsFloat(char *fstring);
```

```
// Функция, преобразующая строку в число типа int
int IsInt(char *istring);
```

Рассмотрим код функции IsFloat(). Он интересен тем, что иллюстрирует многие методы, применяющиеся при синтаксическом анализе.

```
float IsFloat(char *fstring)
{
    // Проверяет, может ли данная строка быть преобразована
    // в число с плавающей запятой. Если да, функция
    // выполняет это преобразование. В противном случае
    // функция возвращает значение FLT_MIN. Проверяется
    // соответствие шаблону
    // [whitespace] [sign] [digits] [.digits]
    // [ {d | D | e | E } [sign] digits]
    char *string = fstring;

    // [whitespace]
    while(isspace(*string)) string++;

    // [sign]
    if (*string=='+' || *string=='-') string++;

    // [digits]
    while(isdigit(*string)) string++;

    // [.digits]
    if (*string=='.')
    {
        string++;
        while(isdigit(*string)) string++;
    }

    // [ {d | D | e | E } [sign] digits]
    if (*string=='e' || *string=='E' ||
        *string=='d' || *string=='D')
    {
        string++;
        // [sign]
        if (*string=='+' || *string=='-') string++;

        // [digits]
        while(isdigit(*string)) string++;
    }

    // Дошли до конца исходной строки
    if (strlen(fstring) == (int)(string - fstring))
        return(atof(fstring));
    else
        return(FLT_MIN);
} // IsFloat
```

Сначала я вообще не собирался говорить о синтаксическом анализе. Сам я его терпеть не могу и стараюсь избегать. Однако не хотелось просто поместить в книгу анализатор, представляющий собой для неискушенного читателя черный ящик, а затем писать

"магические" функции, способные загружать файлы в новом формате. Теперь вы, по крайней мере, ознакомились с этими функциями и получили представление о том, что они делают. Более того, мы создали очень полезные вспомогательные функции! В файле DEMOII8\_7.CPP|EXE содержится демонстрационная программа (снимок экрана, полученного при ее запуске, приведен на рис. 8.46), позволяющая загружать текстовый файл, прогонять его через анализатор и производить поиск по указанным образцам.

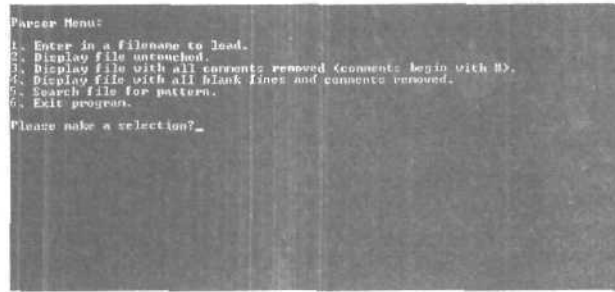


Рис. 8.46. Тестовый запуск анализатора

В каталог, соответствующий данной главе, включены два текстовых файла, TEST1.TXT и TEST2.TXT. Их содержимое также приведено ниже.

```
# test 1
object name: tank
num vertices: 4
```

```
vertexlist:
0 10 20
5 90 10
3 4 3
1 2 3
```

end object # Это комментарий!

Также представляем второй файл.

```
# test 2
This is a Line...
# Закомментировано!
And this a 3rd...
```

Вкратце процесс выглядит так: нужно запустить программу, загрузить файл, указав его имя, а затем с помощью различных предназначенных для анализа функций вывести содержимое файла на экран. Попробуйте применить к файлу TEST1.TXT шаблон, подобный такому: "[i] [i] [i]". Ознакомьтесь с программой; так вы приобретете навыки использования класса-анализатора и вспомогательных функций.

## Формат .ASC 3D Studio MAX

Текстовый формат .ASC от компании 3D Studio Max достаточно прост. В нем содержится информация о каждом многоугольнике, в том числе о его вершинах, и о материале, из которого он сделан. К сожалению, файлы в этом формате не содержат информацию о текстуре и о коэффициентах отражения, поэтому формат .ASC нельзя считать дос-

таточным для хранения информации о полностью текстурированных объектах. Однако этот формат в 10 раз лучше, чем формат .PLG. Кроме того, в него можно экспортировать файлы, созданные в 99% программных продуктов, предназначенных для формирования трехмерных моделей, поэтому этот формат облегчает жизнь. С другой стороны, для многоугольников в формате .ASC из-за его ограничений невозможно выбирать тип затенения. Если вам понадобится больше информации по этому вопросу, повторите главу 6, "Введение в трехмерную графику", в которой описаны различные форматы файлов. Для примера я создал в формате .ASC стандартный куб.

Ambient tight color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Cube"

Tri-mesh, Vertices: 8 Faces: 12

Vertex list:

Vertex 0: X:-1.000000 Y:-1.000000 Z:-1.000000

Vertex 1: X:-1.000000 Y:-1.000000 Z:1.000000

Vertex 2: X:1.000000 Y:-1.000000 Z:-1.000000

Vertex 3: X:1.000000 Y:-1.000000 Z:1.000000

Vertex 4: X:-1.000000 Y:1.000000 Z:-1.000000

Vertex 5: X:1.000000 Y:1.000000 Z:-1.000000

Vertex 6: X:1.000000 Y:1.000000 Z:1.000000

Vertex 7: X:-1.000000 Y:1.000000 Z:1.000000

Face list:

Face 0: A:2 B:3 C:1 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 1: A:2 B:1 C:0 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 2: A:4 B:5 C:2 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 3: A:4 B:2 C:0 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 4: A:6 B:3 C:2 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 5: A:6 B:2 C:5 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 6: A:6 B:7 C:1 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 7: A:6 B:1 C:3 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 8: A:6 B:5 C:4 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Face 9: A:6 B:4 C:7 AB:1 BC:1 CA:1

Material:"r255g255b255a0"

Smoothing: 1

Чтобы читать файлы в этом формате, необходимо создать синтаксический анализатор, **умеющий** выполнять такие действия:

1. считывать имя объекта, а также **количество составляющих** его вершин и многоугольников;
2. считывать список вершин;
3. считывать параметры каждого многоугольника, включая информацию о **цвете** материала в формате RGB.

Ознакомившись с файлами в рассматриваемом формате, легко понять, что в них почти не содержится никакой информации кроме той, которую будет считывать наш анализатор. Он легко справится с этим форматом. Единственный недостаток — то, что невозможно задавать двусторонние многоугольники, создавать для них текстуру, модель освещения и т.п. В основном указывается только цвет каждого многоугольника — и все. В программе, предназначенной для считывания файлов, должна быть предусмотрена возможность работы с такой информацией, как модель освещения. Однако пока что ограничимся программой, **загружающей** файлы в формате **.ASC** в 8-битовом и 16-битовом графических режимах, а затем **заполняющей** соответствующие структуры данных, а также задающей в качестве параметров источников освещения и других подобных характеристик величины, принятые по умолчанию. Прежде чем продемонстрировать, как это делается, я хочу отдельно рассмотреть определение одного многоугольника. Это позволит убедиться в том, что вам все понятно.

```
Face 10:  A:1 B:7 C:4 AB:1 8C:1 CA:1
Material:"r255g255b255a0"
Smoothing: 1
```

Итак, мы рассматриваем поверхность 10. Она состоит из трех многоугольников с вершинами A, B и C (фактически, их номера — 1, 7 и 4, соответственно). Обозначения AB, BC и CA указывают порядок обхода вершин многоугольника (мы эту информацию не используем), а элемент сглаживания (**Smoothing**) отвечает за то, будут ли усредняться нормали многоугольника (также не используется). Более полное описание данного формата можно найти на **прилагающемся** компакт-диске в каталоге 3D\_DOCS\File Formats\.

Это почти **все**, что я хотел сказать о формате **.ASC**. Теперь рассмотрим прототип функции, загружающей файлы в этом формате.

```
int Load_OBJECT4DV1_3DSASC(
    OBJECT4DV1_PTR obj,      // Указатель на объект
    char *filename,          // Имя файла в формате ASC
    VECTOR4D_PTR scale,      // Начальные масштабные
                             // множители
    VECTOR4D_PTR pos,        // Начальное положение
    VECTOR4D_PTR rot          // Начальные углы вращения
    int vertex_flags)        // Флаги, определяющие
                             // порядок обхода вершин
```

Прототип этой функции почти идентичен прототипу функции, загружающей файлы в формате **.PLG**. Отличие состоит в наличии параметра **vertex\_flags**. Он предоставляет пользователю больший контроль над загрузкой объектов. Его необходимость объясняется тем, что в процессе создания объектов с **помощью моделирующих** программ координаты многократно инвертируются или заносятся в буфер и извлекаются из него. При этом порядок обхода вершин может поменяться. Например, наш игровой процессор использует левую систему координат, а программа, в которой создается

модель объекта, может использовать правую. В этом случае нам понадобится поменять направление оси z. Ниже приведен список флагов, к которым можно применять логическую операцию OR.

```
#define VERTEX_FLAGS_INVERT_X 1 // Меняет знак координат X
#define VERTEX_FLAGS_INVERT_Y 2 // Меняет знак координат Y
#define VERTEX_FLAGS_INVERT_Z 4 // Меняет знак координат Z
#define VERTEX_FLAGS_SWAP_YZ 8 // Преобразует правую
#define VERTEX_FLAGS_SWAP_XZ 16 // систему координат в
#define VERTEX_FLAGS_SWAP_XY 32 // левую
#define VERTEX_FLAGS_INVERT_WINDING_ORDER 64 // Меняет
// порядок обхода вершин
```

Ниже представлена загрузка объекта CUBE.ASC при измененном порядке обхода вершин.

```
int Load_OBJECT4DV1_3DSASC(&obj, "CUBE.ASC",
    NULL, NULL, NULL
    VERTEX_FLAGS_INVERT_WINDING_ORDER);
```

**ВНИМАНИЕ**

Программа может работать только с текстовыми файлами в формате .ASC! Не считывайте с ее помощью бинарные файлы, иначе в работе программы произойдет сбой.

Код функции слишком длинный, поэтому здесь он не приводится. Чтобы ознакомиться с ним, откройте файл T3DLIB6.CPP, который содержится на прилагаемом компакт-диске. Большой объем этого файла объясняется тем, что он переполнен фрагментами, в которых производится обработка ошибок (надеюсь, вы обратите на них внимание). Если загружаемые файлы корректны, то ошибки возникать не будут. Обработка ошибок носит иллюстративный характер; кроме того, она поможет вам, если вы решите вручную ввести объект, рожденный вашей фантазией.

В качестве примера использования функции ознакомьтесь с демонстрационной функцией DEMO118\_8.CPP|EXE. На рис. 8.47 приведена копия экрана, полученная при работе этой функции. Упомянутая демонстрационная программа позволяет загружать с диска любой объект в формате .ASC с помощью меню. При этом нужно указывать полный путь к объекту. На прилагаемом компакт-диске в папке CHAPTER8\ содержится несколько созданных мной объектов:

- SPHERE01.ASC (оси Y и Z переставлены, и изменен порядок обхода вершин);
- CAR01.ASC (оси Y и Z переставлены, и изменен порядок обхода вершин);
- HAMMER02.ASC.

**ВНИМАНИЕ**

Нельзя загружать объект, в котором количество вершин превышает максимально допустимое число для процессора, с которым вы работаете. В данный момент для вершин и многоугольников установлены такие максимальные значения:

```
// Задано для объектов версии 1
#define OBJECT4DV1_MAX_VERTICES 1024
#define OBJECT4DV1_MAX_POLYS 1024
```

Эти определения, конечно же, содержатся в модуле библиотеки T3DLIB5.H.

Кроме того, в диалоговом окне, предназначенном для загрузки файла, есть поля, позволяющие менять направление координатных осей y и z, а также порядок обхода вершин. Возможно, вам придется подбирать параметры, чтобы добиться правильной загрузки объекта.

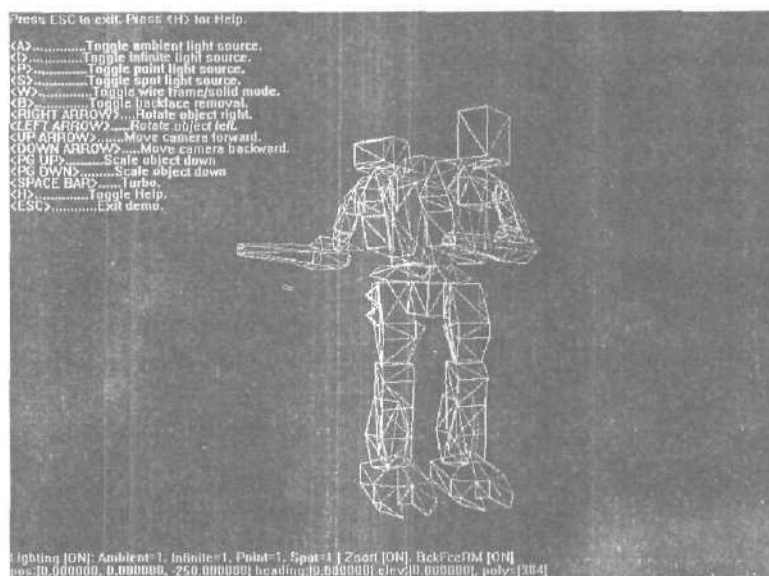


Рис. 8.47. Загрузчик файлов в формате 3D Studio .ASC в действии

## Текстовый формат .COB trueSpace

После длительных раздумий и некоторых компромиссов я пришел к выводу, что лучшим среди представленных на рынке статических форматов, не считая формата Microsoft .X, является .COB. Он очень надежный и поддерживает все особенности, с которыми приходится иметь дело при моделировании материалов. Таким образом, в этом формате можно выбирать вид затенения, указывая материал, из которого изготовлены многоугольники. Кроме того, формат .COB поддерживает текстуру и обладает многими другими возможностями. Если вы не используете trueSpace, то для работы с этим форматом вам потребуется соответствующий конвертор — такой, как Polytrans. Что же касается формата .3DS, то я просто отказываюсь писать программу для чтения таких файлов. Этот формат — настоящее бедствие, я не могу терпеть запутанных и устаревших форматов. Поэтому если вы хотите работать с форматом .3DS — пожалуйста, дерзайте. Предлагаю вам самостоятельно написать программу, считывающую файлы в этом формате. Честно говоря, не такой уж он и плохой...

В главе 6, "Введение в трехмерную графику", обсуждался формат .COB (его подробный анализ содержится на прилагаемом компакт-диске в папке 3D\_DOCS\File Formats\). Однако посмотрим еще раз на текстовый дамп объекта в формате .COB. Это файл CALIGARI\_COB\_FORMAT\_EXAMPLE01.COB, а на рис. 8.48 приведено изображение содержащегося в нем объекта.

Этот объект представляет собой куб в формате trueSpace, на одной из граней которого нанесена текстура. Ниже приведены данные в текстовом виде. Сюда в целях более удобного чтения добавлены пустые строки, но на самом деле их там быть не должно. Кроме того, некоторые интересные нас фрагменты выделены полужирным шрифтом, а определенные сложные разделы, имеющие отношение к макетам и светимости, выброшены. Поэтому если вы хотите ознакомиться со всеми особенностями объекта, обратитесь к исходному файлу. Он слишком длинный, поэтому мы не стали приводить его здесь полностью.

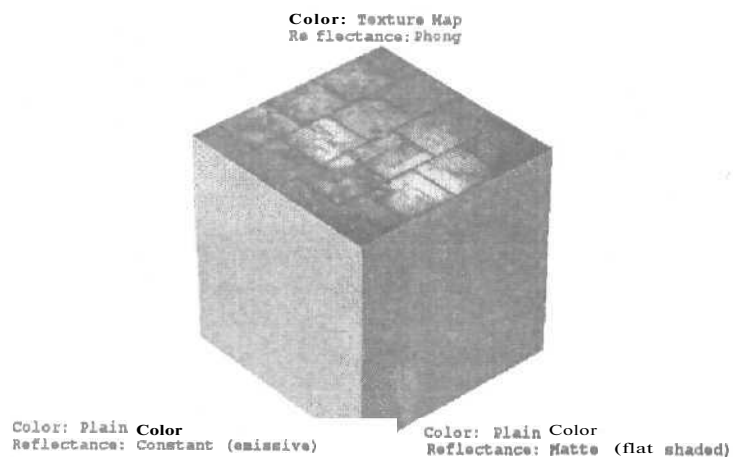


Рис. 8.48. Модель куба с разными типами поверхностей

Caligari V00.01ALH

PolH V0.08 Id 18661892 Parent 0 Size 00001060

Name Cube,l

center 0 0 0

x axis 1 0 0

y axis 0 1 0

z axis 0 0 1

Transform

1 0 0 1.19209e-007

0 1 0 0

0 0 1 -1

0 0 0 1

World Vertices 8

1.000000 -1.000000 0.000000

1.000000 -1.000000 2.000000

1.000000 -1.000000 0.000000

1.000000 -1.000000 2.000000

1.000000 1.000000 0.000000

1.000000 1.000000 0.000000

1.000000 1.000000 2.000000

1.000000 1.000000 2.000000

Texture Vertices 6

0.000000 0.000000

0.000000 1.000000

0.000000 0.000000

0.000000 1.000000

1.000000 0.000000

1.000000 1.000000

Faces 12

Face verts 3 flags 0 mat 4

<0,0> <1,1> <3,5>  
 Face verts 3 flags 0 mat 4  
 <0,0> <3,5> <2,4>  
 Face verts 3 flags 0 mat 0  
 <0,1> <2,5> <5,4>  
 Face verts 3 flags 0 mat 0  
 <0,1> <5,4> <4,0>  
 Face verts 3 flags 0 mat 1  
 <2,2> <3,3> <6,5>  
 Face verts 3 flags 0 mat 1  
 <2,2><6,5><5,4>  
 Face verts 3 flags 0 mat 0  
 <1,0> <7,1> <6,5>  
 Face verts 3 flags 0 mat 0  
 <1,0> <6,5> <3,4>  
 Face verts 3 flags 0 mat 2  
 <4,4> <5,0> <6,1>  
 Face verts 3 flags 0 mat 2  
 <4,4> <6,1> <7,5>  
 Face verts 3 flags 0 mat 3  
 <0,4> <4,2> <7,3>  
 Face verts 3 flags 0 mat 3  
 <0,4> <7,3> <1,5>  
**Mat1 V0.06 Id 18659348 Parent 18661892 Size 00000102**  
**mat# 1**  
 shader: phong facet: auto32  
 rgb 0.00784314,1,0.0352941  
**alpha 1 ka 0.1 ks 0.1 expo ior 1**  
 ShBx V0.03 Id 18659349 Parent 18659348 Size 00000383  
**Shader class: color**  
**Shader name: "plain color" (plain) p- this is an example**  
**↳ of a non-textured material**  
 Number of parameters: 1  
 colour: color (2, 255,9)  
 Flags: 3  
 Shader class: transparency  
 Shader name: "none" (none)  
 Number of parameters: 0  
 Flags: 3  
**Shader class: reflectance**  
**Shader name: "constant" (constant) p this material**  
**↳ is constant shaded, or emmissive to us**  
 Number of parameters: 0  
 Flags: 3  
 Shader class: displacement  
 Shader name: "none" (none)  
 Number of parameters: 0  
 Flags: 3  
  
**Mat1 V0.06 Id 18658628 Parent 18661892 Size 00000085**  
**mat# 2**  
 shader: phong facet: auto32

rgb 1,0,0  
**alpha 1 ka 0.1 ks 0.1 exp 0 ior 1**  
 ShBx V0.03 Id 18658629 Parent 18658628 Size 00000427  
 Shader **class: color**  
**Shader name: "plain color" (plain) p again a plain**  
 colored material  
 Number of parameters: 1  
 colour: color (255, 0, 0)  
 Flags: 3  
 Shader class: transparency  
 Shader name: "none" (none)  
 Number of parameters: 0  
 Flags: 3  
**Shader class: reflectance**  
**Shader name: "matte" (matte) p this material is fiat**  
 shaded in our engine  
 Number of parameters: 2  
 ambient factor: float 0.1  
 diffuse factor: float 1  
 Rags: 3  
 Shader class: displacement  
 Shader name: "none" (none)  
 Number of parameters: 0  
 Flags: 3  
  
**Mat1 V0.06 Id 18657844 Parent 18661892 Size 00000101**  
**mat# 3**  
 shader: phong facet: auto32  
 rgb 0.0392157,0.0117647,1  
**alpha 1 ka 0.1 ks 0.5 exp 0 ior 1**  
 ShBx V0.03 Id 18657845 Parent 18657844 Size 00000522  
 Shader **class: color**  
**Shader name: "plain color" (plain) p again a colored material**  
 Number of parameters: 1  
 colour: color (10, 3, 255)  
 Flags: 3  
 Shader class: transparency  
 Shader name: "none" (none)  
 Number of parameters: 0  
 Flags: 3  
**Shader class: reflectance**  
**Shader name: "plastic" (plastic) P this means**  
 use gouraud shading  
 Number of parameters: 5  
 ambient factor: float 0.1  
 diffuse factor: float 0.75  
 specular factor: float 0.5  
 roughness: float 0.1  
 specular colour: color (255, 255, 255)  
 Flags: 3  
 Shader class: displacement  
 Shader name: "none" (none)

Number of parameters: 0

Flags: 3

**Mat1 V0.06 Id 18614788 Parent 18661892 Size 00000100**

**mat# 4**

shader: phong facet: auto32

**rgbl,0.952941,0.0235294**

**alpha 1 ka 0.1 ks 0.1 exp 0 ior 1**

ShBx V0.03 Id 18614789 Parent 18614788 Size 00000515

**Shader class: color**

**Shader name: "plain color" (plain) p a plain colored material**

Number of parameters: 1

colour: color (255, 243, 6)

Flags: 3

Shader class: transparency

Shader name: "none" (none)

Number of parameters: 0

Flags: 3

**Shader class: reflectance**

**Shader name: "phong" (phong) P actually use a phong**

↳ **shader in our engine**

Number of parameters: 5

ambient factor: float 0.1

diffuse factor: float 0.9

specular factor: float 0.1

exponent: float 3

specular colour: color (255, 255, 255)

Flags: 3

Shader class: displacement

Shader name: "none" (none)

Number of parameters: 0

Flags: 3

**Mat1 V0.06 Id 18613860 Parent 18661892 Size 00000182**

**mat# 0**

shader: phong facet: auto32

**rgb 1,0.952941,0.0235294**

**alpha 1 ka 0.1 ks 0.1 exp 0 ior 1**

**texture: 36D:\Source\models\textures\wall01.bmp**

offset 0,0 repeats 1,1 flags 2

ShBx V0.03 Id 18613861 Parent 18613860 Size 00000658

**Shader class: color**

**Shader name: "texture map" (caligari texture) β this**

↳ **means this material is a texture map**

Number of parameters: 7

**file name: string "D:\Source\models\textures\wall01.bmp"**

↳ **β here the texture**

S repeat: float 1

T repeat: float 1

S offset: float 0

T offset: float 0

animate: bool 0

```

filter: bool 0
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shadername: "phong" (phong) β the shader for this
material should be phong
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.9
specular factor: float 0.1
exponent: float 3
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
END V1.00 Id 0 Parent 0 Size 0

```

Этот формат можно считать вполне удобочитаемым. Опустим некоторые подробности и приведем общее описание процесса разработки программы, считывающей файлы в формате .COB.

1. Считываем заглавную строку, расположенную сверху файла.
2. Считываем строки с именем объекта, координатами его центра, а также строки, соответствующие осям  $x$ ,  $y$  и  $z$ . При этом предполагается, что объект задан в локальной системе координат. Это преобразование необходимо применить к вершинам объекта, поскольку все объекты в формате trueSpace экспортируются без преобразования вершин. Это делается, чтобы в результате дополнительных операций не терялась точность. Другими словами, когда в trueSpace производится вращение или сдвиг куба, то программа, с **помощью** которой создается модель, записывает эти преобразования и отображает их на экране, однако единичный куб остается при этом неизменным, чтобы не терялась точность.
3. Считываем строку с надписью Transform (преобразовать) и матрицу, которая следует после нее. С помощью этой матрицы задается переход из локальной системы отсчета в глобальную. В большинстве случаев это единичная матрица, но мы, как и в предыдущей программе, реализуем поддержку преобразований. При этом можно не сомневаться, что мы получаем именно ту модель, которая создавалась в программесоздателе моделей, причем сохраняем правильную ориентацию в пространстве.
4. Считываем строку с надписью World Vertices, а затем — список вершин в котором координаты заданы в порядке  $(x, y, z)$ .
5. Считываем строку с надписью Texture Vertices, а затем — координаты  $\langle u, v \rangle$  каждой текстуры. Здесь применяется соглашение, позволяющее несколько сократить объем файла. Обычно для определения каждого треугольника, входящего в состав текстуры, требуется задать координаты трех точек. Однако в формате .COB координаты **точек**, общих для двух треугольников, не повторяются. Таким образом, если имеется текстура, состоящая из 100 треугольников, то в большинстве случаев в разделе

Texture Vertices будет не 300 координат, как можно было бы ожидать, а немного меньше. Впрочем, иногда координаты могут и дублироваться, хотя в этом нет необходимости. Мне так и не удалось понять, какая закономерность лежит в основе этого, но, независимо от этой закономерности, будем просто считывать координаты в массив, а затем помещать их в массив с конечными координатами текстуры (у нас он пока что не задан, но не стоит об этом беспокоиться).

#### СОВЕТ

Для тех, кто **незнаком** с координатами текстуры, заметим, что это просто точки текстуры, **соответствующие** вершинам многоугольников, из которых состоит объект. В большинстве случаев текстуры имеют такую систему координат, как показано на рис. 8.49. Координата  $u$  соответствует оси  $x$ , а координата  $v$  — оси  $y$ . Кроме того, 99% всех текстур имеют квадратную форму. Таким образом, чтобы отобразить текстуру, приведенную на рис. 8.49, на пару треугольников, следует использовать координаты, как показано на этом рисунке. Далее в книге этот вопрос обсуждается **подробнее**.

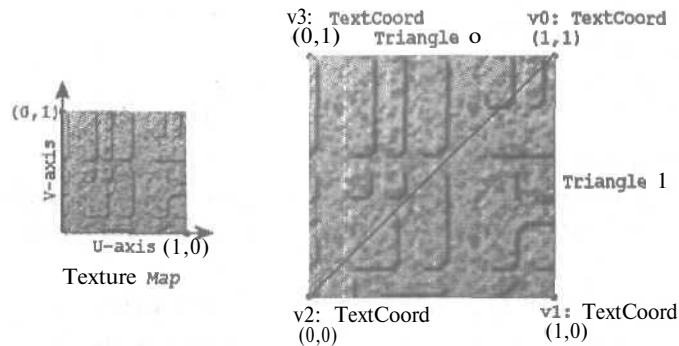


Рис. 8.49. Стандартные координаты текстуры

6. Считываем строку с надписью **Faces**, а затем — данные, **описывающие** каждую грань. Грани могут состоять из большого числа вершин, однако мы должны убедиться, что все экспортированные ячейки имеют по три вершины. Таким образом, строки с данными о гранях должны выглядеть как

```
Face verts 3 ftag sj^f mat mm
```

где **ff mm** — флаг и номер материала, соответственно. Флаг нас не интересует, поэтому не будем обращать на него **внимания**, а вот номер материала играет важную роль. Это номер (отсчитываемый от нуля), который указывает, какой материал следует приписать данной грани. В **процессе** синтаксического разбора этот материал следует запомнить с тем, чтобы при дальнейшем считывании данных о материалах можно было выбрать нужный.

Далее следует информация о самой грани, представляющая собой набор индексов вершин в формате  $\langle v_1, t_1 \rangle \langle v_2, t_2 \rangle \dots$ . Первый элемент — это индекс вершины треугольника, а второй — координата текстуры. Таким образом, запись  $\langle 1, 2 \rangle \langle 9, 5 \rangle \langle 3, 8 \rangle$  означает, что данный треугольник состоит из вершин 1, 9, 3, и что координаты текстуры для этих вершин — 2, 5, 8, причем между вершинами и координатами текстуры существует взаимно однозначное соответствие.

7. Начинается неприятная часть, Нужно считать данные о материалах, причем они могут следовать не по порядку. Наш игровой процессор не будет похож на **аппа-**

ратный процессор HALO, поэтому в нем не будут поддерживаться все особенности материалов. Нам каким-то образом нужно закодировать следующие свойства.

```
#define MATV1_ATTR_2SIDED          0x0001
#define MATV1_ATTR_TRANSPARENT    0x0002
#define MATV1_ATTR_8BITCOLOR      0x0004
#define MATV1_ATTR_RGB16          0x0008
#define MATV1_ATTR_RGB24          0x0010

#define MATV1_ATTR_SHADE_MODE_CONSTANT 0x0020
// Синоним
#define MATV1_ATTR_SHADE_MODE_EMISSIVE 0x0020
#define MATV1_ATTR_SHADE_MODE_FLAT    0x0040
#define MATV1_ATTR_SHADE_MODE_GOURAUD 0x0080
#define MATV1_ATTR_SHADE_MODE_FASTPHONG 0x0100
#define MATV1_ATTR_SHADE_MODE_TEXTURE 0x0200
```

Предполагается, что нам не придется работать с двусторонними поверхностями. Очень мало программ, предназначенных для создания трехмерных моделей, поддерживают концепцию двусторонних поверхностей в списке многоугольников или материалов, поэтому ее необходимость весьма спорная. Кроме того, все материалы в таких программах задаются в пространстве RGB, поэтому понятие глубины цвета представляется искусственным; этот параметр вычисляется в процессе загрузки на основе того режима, в котором загружаются модели. Единственное, чем мы можем управлять, — это свойства, выделенные полужирным шрифтом. Преимуществом формата .COB является то, что в нем поддерживается ряд особенностей материалов, с помощью которых можно указывать, какую модель освещения следует применить к тому или иному многоугольнику.

Например, мы моделируем корабль и хотим, чтобы его двигатель затусевывался с помощью постоянного затенения, а все остальное — по Гуро. Для этого нужно применить материал, из которого можно было бы извлечь такую информацию. Если программа, в которой создаются модели, не поддерживает экспортирование информации такого рода, нам придется проявить большую смекалку. Например, можно сделать предположение, что все многоугольники, окрашенные в серый цвет, излучают, а их цвет представляет собой оттенки серого цвета, интенсивность которых пронумерована от 0 до 15.

К счастью, нет нужды прибегать к таким головоломным трюкам, поскольку в программе, предназначенной для создания моделей, подобные свойства материала кодируются естественным путем. Однако нельзя сказать, что между режимом затенения в моделирующей программе и тем режимом, который нужно применить, существует взаимно однозначное соответствие. К этому вопросу мы вернемся чуть позже, а теперь поговорим о том, что закодировано в файле. Какая бы ни применялась программа для создания модели, но если мы экспортировали объект в формат trueSpace 4+ .COB, все будет хорошо. Кодирование происходит так, как описано ниже.

Во-первых, производится поиск материала. Данные о материале начинаются строками, похожими на приведенные ниже (вторая строка — номер материала).

```
Mat1 V0.06 Id 142522724 Parent 18646020 Size 00000182
mat# 0
```

Выше приведено определение материала с номером 0. Поэтому его следует применить ко всем считанным многоугольникам, в которых был указан этот номер материала. Следующие интересующие нас строки имеют следующий вид.

```
rgb 0.0392157,0.0117647,1
alpha 1 ka 0.1 ks 0.5 exp 0 ior 1
```

В строке, которая начинается надписью `rgb`, записан цвет материала. Сама надпись `rgb` лишняя, причем встречается она неоднократно. В следующей строке, которая начинается надписью `alpha`, заданы такие параметры, как прозрачность материала, коэффициенты отражения и другие. Все они перечислены в табл. 8.4.

**Таблица 8.4. Кодировка свойств материалов в формате .COB**

Переменная	Значение	Диапазон
<code>alpha</code>	Прозрачность	0 - 1
<code>ka</code>	Коэффициент <b>общего</b> отражения	0 - 1
<code>ks</code>	Коэффициент зеркального отражения	0 - 1
<code>exp</code>	Показатель степени освещения	0 - 1
<code>ior</code>	Коэффициент преломления	(в нашей модели не используется)

А теперь поговорим о модели затенения. Если вы посмотрите на определение каждого материала в списке, то увидите, что некоторые строки выделены полужирным шрифтом. Это выглядит следующим образом.

**Shader class: color**

**Shader name: "plain color" (plain)**

•  
•  
•  
•

**Shader class: reflectance**

**Shadername: "phong" (phong)**

•  
•

Внимательно посмотрим на эти строки. Строки **Shader class: color** и **Shader class: reflectance**— это ключ к декодированию информации о виде затенения. Из них можно понять, является ли многоугольник одноцветным (в таком случае у него излучательное затенение, плоское затенение, затенение по Гуро или по Фонгу) или обладает текстурой.

Во-первых, нужно получить сведения о модели затенения. Это не так просто; следует найти строку наподобие следующей.

Shader class: **color**

После нее следует строка наподобие

**Shader name: "xxxxxx" (xxxxxx)**

где вместо последовательности символов `xxxxxx` будут надписи **"plain color"** и **"plain"** для одноцветных многоугольников или надписи **"texture map"** и **"caligari texture"** для текстур. Далее необходимо найти строку

Shader class: **reflectance**

в которой закодирован вид затенения. После этого ищем строку

Shader name: **"xxxxxx" (xxxxxx)**

Согласно содержащейся в ней информации и **данным**, приведенным в табл. 8.5, можно сделать вывод о системе затенения.

**Таблица 8.5. Выбор вида затенения в процессоре, обрабатывающем файлы в формате .COB**

<i>Модель затенения в формате .COB</i>	<i>Модель затенения в разрабатываемом процессоре</i>
"constant"	MATV1_ATTR_SHADE_MODE_CONSTANT
"matte"	MATV1_ATTR_SHADE_MODE_FLAT
"plastic"	MATV1_ATTR_SHADE_MODE_GOURAUD
"phong"	MATV1_ATTR_SHADE_MODE_FASTPHONG
"texture map"	MATV1_ATTR_SHADE_MODE_TEXTURE

**НА ЗАМЕТКУ**

Отображение текстуры необходимо объединить с помощью побитового ИЛИ с одним из предыдущих режимов, чтобы текстура также затенялась.

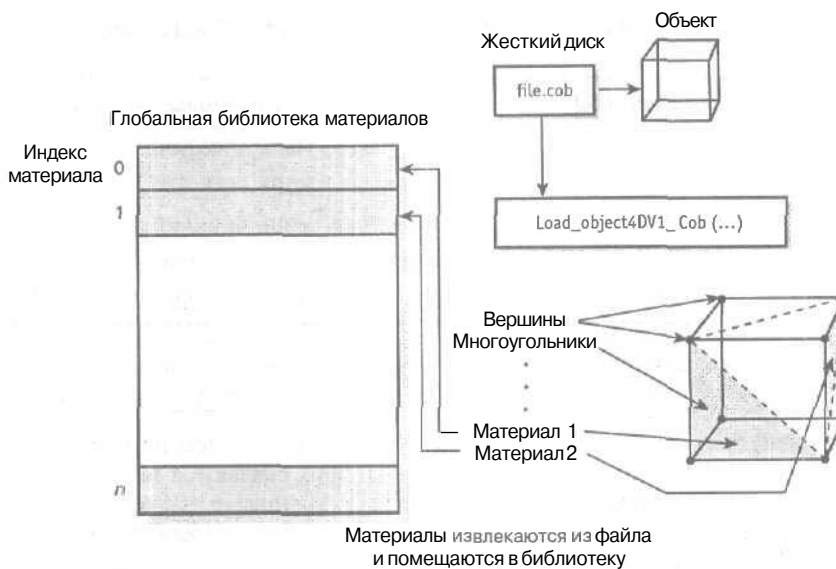
Как все запутано, правда? На самом деле задача проще, чем кажется. Рассмотрим еще одну проблему. У программ, с помощью которых создаются модели, нет средств, чтобы экспортировать информацию о виде модели. Например, вы хотите создать игровой мир с помощью программы 3D Max. Но как при этом вы укажете, что дверь — это дверь? Понимаете, в чем проблема? Здесь ситуация аналогична предыдущей: проблема не так страшна, но она существует. Нужно найти способы закодировать в файле информацию о виде модели.

А теперь, когда я совсем вас запутал, вспомним, как с помощью программы trueSpace создаются модели в формате .COB, которые способна считывать наша программа. Сначала происходит процесс моделирования объекта, а затем необходимо назначить ему материалы. Если нужно, чтобы многоугольник был одноцветным, напротив поля с цветом материала остается значение "plain color". Далее в той части модели, в которой описываются коэффициенты отражения, выбирается модель отражения. Она может соответствовать постоянному, матовому, пластиковому затенению или затенению по Фонгу, в зависимости от того, чего именно вы хотите добиться от процессора. Пока что будут поддерживаться только постоянное и плоское затенения. Если же нужно, чтобы поверхность обладала текстурой, следует текстурировать ее обычным образом; при этом в поле "Shader name" появится надпись "texture". После этого нужно будет задать модель отражения, которую следует применить к текстуре. Пока что наш игровой процессор не поддерживает текстуры, поэтому не пытайтесь их применять.

Теперь, когда рассмотрены все особенности формата .COB, поговорим о материалах. Фактически во время загрузки объекта в этот формат данные о материалах **вносятся** в специальную библиотеку (рис. 8.50). Однако при этом возникают некоторые проблемы. Во-первых, наш игровой процессор пока что не поддерживает материалов; он поддерживает только источники освещения. Во-вторых, материал, который загружается для одного объекта, может одновременно использоваться другим объектом. Но, как уже было сказано, материалы у нас пока что не поддерживаются, поскольку у нас нет формата трехмерных изображений, поддерживающего материалы.

**ВНИМАНИЕ**

Объекты в формате .COB должны состоять только из треугольников, поэтому перед экспортированием объектов в этот формат их следует разбить на треугольники. В целях экономии пространства в предыдущем примере в качестве элементарных участков поверхности выступают четырехугольники, однако разрабатываемый нами игровой процессор воспринимает только треугольники.



**Рис. 8.50. Программа, считывающая файлы в формате .COB, загружает данные о материалах в глобальную библиотеку**

Теперь, когда вы получили некоторое представление о формате .COB, рассмотрим приведенный ниже прототип загрузчика файлов. Он почти идентичен прототипу загрузчика файлов в формате .ASC, за исключением того, что обладает несколькими дополнительными возможностями.

```
int Load_OBJECT4DV1_COB(
    OBJECT4DV1_PTR obj,    // Указатель на объект
    char *filename,        // Имя файла формата Caligari COB
    VECTOR4D_PTR scale,    // Начальный масштабный множитель
    VECTOR4D_PTR pos,      // Начальное положение
    VECTOR4D_PTR rot,      // Начальные углы вращения
    int vertex_flags);     // Флаг, отвечающий за изменение
                          // порядка обхода вершин и
                          // преобразований
```

Единственное различие между упомянутыми выше функциями — добавление пары флагов `vertex_flags`, которые можно объединять с помощью побитового оператора ИЛИ, чтобы активизировать преобразование в локальную или глобальную систему координат, которые уже описывались для данного формата.

```
#define VERTEX_FLAGS_TRANSFORM_LOCAL    512
// Если формат файла предусматривает переход в локальную
// систему координат, следует его выполнить!
```

```
#define VERTEX_FLAGS_TRANSFORM_LOCAL_WORLD 1024
// Если формат файла предусматривает переход из локальной в
// глобальную систему координат, следует его выполнить!
```

Ниже приведен пример загрузки типичного объекта в формате .COB, в котором флаги установлены так, как это делается большинством моделирующих программ при экспорте объектов.

```
Load_OBJECT4DV1_COB(&obj_tank,
    "D:/Source/models/cobs/hammer03.cob",
    &vscale, &vpos, &vrot
    VERTEX_FLAGS_SWAP_YZ[
    VERTEX_FLAGS_TRANSFORM_LOCAL]);
```

Из этого кода видно, что в функцию передается коэффициент масштабного преобразования, положение объекта и векторы его вращения. Конечно, эти параметры не являются обязательными.

Завершим на этом описание формата **.COB**. Как я уже говорил, полное его описание содержится на прилагаемом компакт-диске в каталоге 3D\_DOCS\File Formats\ . Кроме того, полезно изучить код, содержащийся в файлах T3DLIB6.CPP[H. В качестве примера использования загрузчика ознакомьтесь с программой DEMO118\_9.CPP[EXE; она идентична программе-загрузчику файлов в формате **.ASC**, но работает с файлами в другом формате.

Кроме того, в диалоговом окне, которое появляется на экране после активизации команды меню **File⇒Load**, имеются флажки, позволяющие менять направления осей *y* и *z* и порядок обхода вершин, а также переходить из локальной системы координат в глобальную. Возможно, придется подбирать эти параметры, чтобы объект загрузился корректно. Наконец, напомним, что в программе-загрузчике есть место только для 1024 вершин и 1024 многоугольников объекта! Вот некоторые объекты в формате **.COB**, которые можно загрузить:

- **SPHERE01.COB** (можно менять направления осей *y* и *z* и порядок обхода вершин);
- **CAR01.COB** (можно менять направления осей *y* и *z*, а также переходить в локальную и глобальную систему координат);
- **HAMMER03.COB** (можно менять направления осей *y* и *z*, а также переходить в локальную и глобальную систему координат).

#### ВНИМАНИЕ

При создании файлов в формате **.COB** следует помнить о том, что поверхности всех объектов **НЕОБХОДИМО** разбивать на треугольники и что программа, которая считывает такие файлы, способна считывать только единичные объекты. Кроме того, убедитесь, что каждый многоугольник задан как одноцветный (т.е. установлен флаг цвета "plane color") и задано плоское затенение (установлен флаг затенения "matte"). Пока что у нас поддерживаются только эти параметры.

## Первое знакомство с бинарным форматом Quake II .MD2

Формат *Quake II* .MD2 — один из наиболее популярных форматов, предназначенных для трехмерной анимации. Этот формат разработан для игры *Quake II* (следовательно, он предназначен для игр), поэтому в таких программах, предназначенных для создания моделей, как 3D Max и trueSpace, он не поддерживается. Для создания моделей в этом формате есть другие инструменты, но к ним мы еще вернемся. А пока я хочу, чтобы вы поняли, какие возможности поддерживает этот формат. По сути, файл в формате **.MD2** представляет собой набор вершин, анимационных кадров, скинов и еще пары вещей, играющих важную роль в игровых моделях *Quake II*. Каждый файл — это заверченный анимационный эпизод для персонажа игры *Quake II*. В таких файлах содержатся отдельные фрагменты, в которых может участвовать этот персонаж (всего 384 кадров): как он ходит, умирает, стреляет, поворачивается и т.д.

**Скины** — это, по сути, текстурные отображения квадратной формы, как правило, представляющие собой файлы в формате **.eps** размером 256x256. Эти файлы используются для текстурирования моделей. Все текстуры находятся в одном и том же скине, а нужная текстура вырезается с помощью текстурных координат. На рис. 8.51 и 8.52 приведены примеры скинов, применяющихся в каркасах в формате **.MD2**.

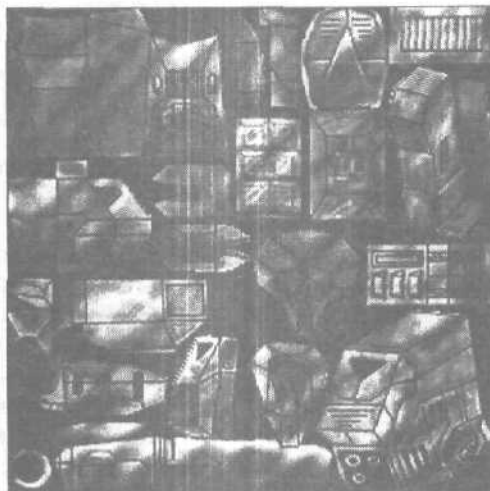


Рис. 8.51. Скин, используемый в формате Quake II .MD2

На рис. 8.52 приведен каркас модели в формате .MD2 (в том виде, в котором он выводится в окне программы Milkshape 3D), загруженный из Web-узла <http://www.planetquake.com/polycount>. Этот Web-узел — замечательный ресурс для получения моделей. Конечно же, если вы планируете использовать эту или любую другую модель в коммерческой игре, вам понадобится получить на это разрешение.

В этом формате поддерживаются также несложные схемы сжатия информации. Этот формат прост для анализа и понимания. Он, конечно же, полностью бинарный, но нам придется смириться с этим. Мы обязательно воспользуемся данным форматом позже, когда перейдем к анимации. Единственный его недостаток состоит в том, что он не поддерживает такого понятия, как материалы, поэтому при загрузке объекта будет применяться материал по умолчанию, чтобы разработанный нами игровой процессор освещения знал, что делать с этой моделью.

## Обзор программных инструментов для создания трехмерных моделей

Прежде чем перейти к следующей главе, я хочу представить краткий обзор некоторых программных инструментов, которые я считаю полезными для моделирования и преобразования форматов, о которых шла речь в этой главе (ознакомительные версии некоторых из них содержатся на прилагаемом компакт-диске).

- **Caligari trueSpace 4.0+** (программа для создания моделей). Эта программа показана на рис. 8.53. Это один из моих любимых программных инструментов — быстрый, удобный в работе и недорогой. Кроме того, он позволяет экспортировать созданные модели в различные форматы. Более подробную информацию об этой программе можно найти на Web-узле <http://www.caligari.com>.
- **3D Studio Max 3.0+** (программа для создания моделей). Эта программа показана на рис. 8.54. Этот программный инструмент в рекламе не нуждается. Слов для его описания нет — есть только выражения :) Он используется для создания игр, фильмов и всего, чего угодно. Конечно же, цена (2500–5000 долл.) может пока-

заться слишком высокой. Более подробную информацию об этой программе можно найти на Web-узле <http://www.discreet.com/products/3dsmax/>.

- **Gmax** — эта программа похожа на 3D Studio Max. Ее окно изображено на рис. 8.54. Этот программный продукт представляет определенный интерес. На самом деле это просто бесплатная, но несколько урезанная версия программы 3D Studio Max. Эту программу можно зарегистрировать как средство для создания игр. В любом случае необходимые сведения можно найти на Web-узле компании Discreet по адресу <http://www.discreet.com/products/gmax/>.

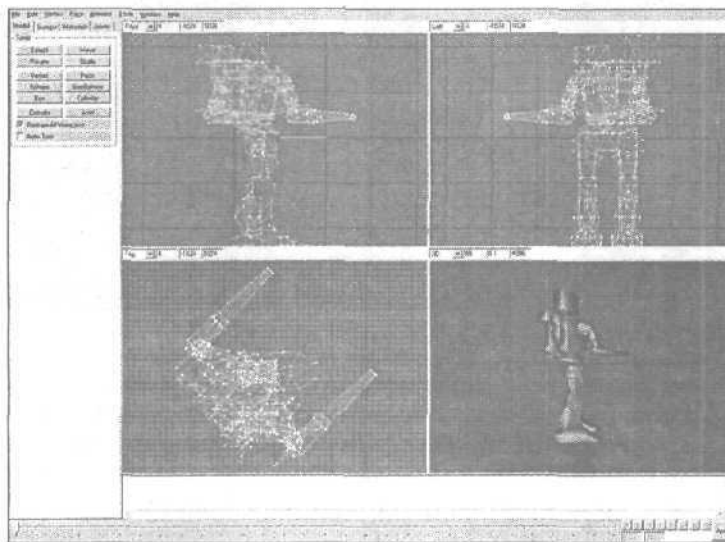


Рис. 8.52. Единичный кадр в формате Quake II .MD2

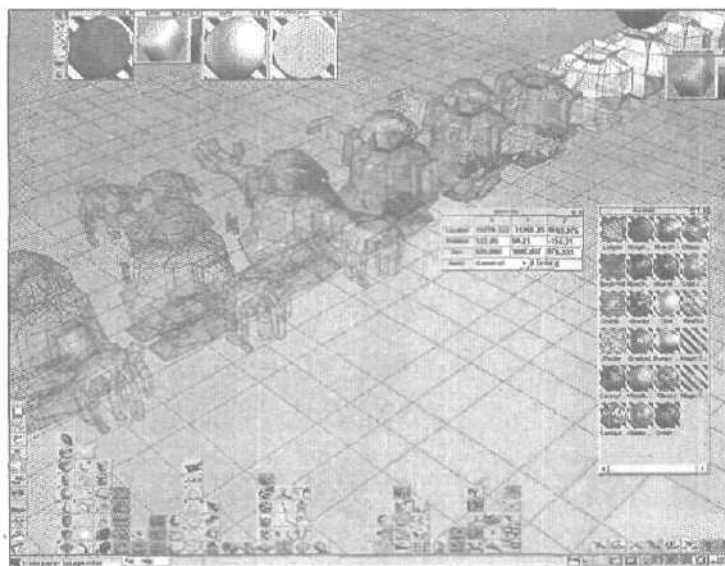


Рис. 8.53. Программа trueSpace в действии

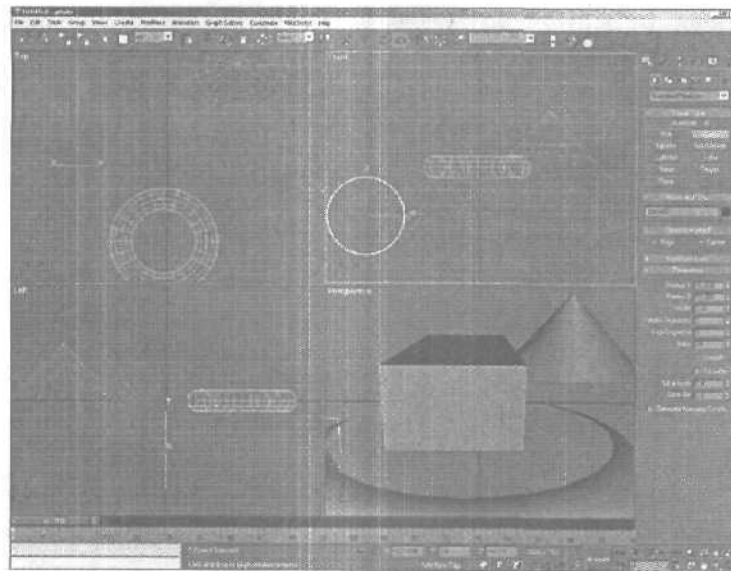


Рис. 8.54. Программа *Gmax Game Studio* в действии

- **Blender** (программа для создания моделей). Эта программа показана на рис. 8.55. Это бесплатная программа, но, судя по тому, что компания NaN временно удалась от дел, мне кажется, что идея бесплатного распространения программ с последующим платным техническим обслуживанием в реальном мире не работает. Несмотря на это, сама программа замечательна. Искренне надеюсь, что ее разработчик сможет лицензировать ее или переделать в реальный программный продукт с тем, чтобы заняться продажей. Более подробную информацию об этой программе можно найти на Web-узле <http://www.blender.nl>.

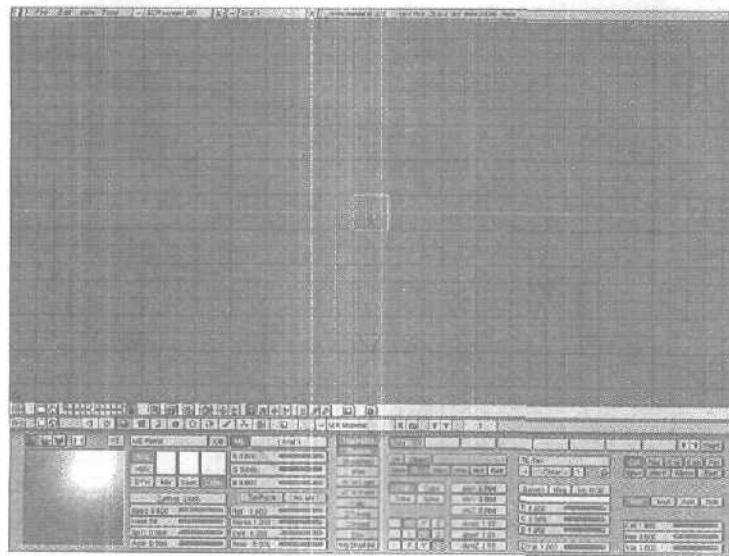


Рис. 8.55. Программа *Blender* в действии

- 

- **Polytrans** (программа для просмотра и экспортирования моделей). У этого программного продукта наиболее полная система импортирования и экспортирования форматов из всех **существующих**. Хорошо налажена поддержка всех трехмерных форматов, однако игровые форматы поддерживаются слабо. Web-узел этой программы находится по адресу <http://www.okino.com/conv/conv.htm>.
- **WorldCraft Editor** (редактор уровней). Эта программа показана на рис. 8.57. С помощью этого программного инструмента созданы миллионы моделей и многие игры, например *Half-Life*. Программу можно использовать для создания игровых миров, которые затем будут импортированы в ваш игровой процессор. На данном этапе нам такой инструмент не нужен, однако когда мы перейдем к разработке эпизодов, действие которых разворачивается в помещении, он окажется полезным. В Internet много ресурсов, посвященных программе WorldCraft, так что вы можете сами их поискать, однако довольно стабильным является Web-узел, расположенный по адресу <http://www.valve-erc.com/>.

Приведем несколько полезных ссылок на Web-узлы, посвященные созданию трехмерных моделей:

- <http://www.planetquake.com/polycount/>
- <http://www.web3d.org/vrml/vft.htm>
- <http://www.gmi.edu/~jsinger/free3dmodels.htm>

Для создания моделей я обычно пользуюсь программами trueSpace и 3D Max, а преобразования форматов и другие манипуляции произвожу в Deep Exploration. Однако, по моему, Milkshape 3D на самом деле обладает намного более мощными возможностями, чем кажется на первый взгляд, и я хотел бы с ним еще немного поэкспериментировать.

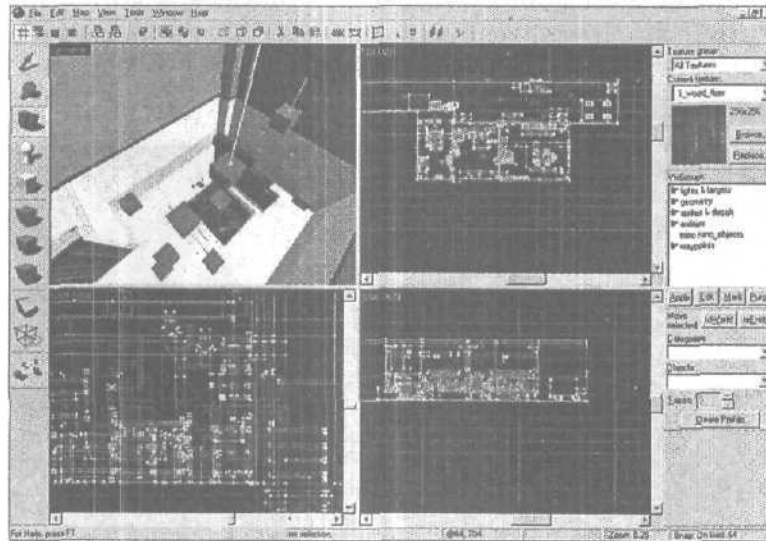


Рис. 8.57. Программа WorldCraft Editor в действии

## Резюме

Эта глава далась мне с трудом! Теоретические исследования сочетались в ней с практической реализацией. Глава содержит множество материала: основы теории света, материалов, источников **освещения**, немного математики, вопросы затенения, сортировки по глубине, описание новых форматов моделей и многое другое. Позже у нас еще будет время освоить все это детальнее. Давайте перейдем к следующей главе и разберемся с более сложными видами затенения, которые выполняются на уровне отдельных пикселей. Кроме того, мы ознакомимся с **текстурированием** и еще раз подробно обсудим растеризацию **треугольников**. К концу следующей главы у нас будет заверченный игровой процессор трехмерных изображений с возможностью создания текстурных отображений и освещения (правда, только для сцен на открытом пространстве).

# ГЛАВА 9

## Интерполяционные методы затенения и аффинное отображение текстур

### В этой главе...

• Особенности нового трехмерного процессора	750
• Обновление и разработка структур трехмерных данных	751
• Новые версии загрузчиков	780
• Обзор растеризации многоугольников	805
• Реализация затенения по Гуро	823
• Основы теории дискретизации	849
• Обновление процессора освещения и растеризации для работы с текстурами	861
• Вопросы оптимизации для 8- и 16-битовых режимов	869
• Итоговые демонстрационные программы	872

Лишь совсем недавно мне удалось дописать все программы для данной главы! Пришлось изменить, написать и переработать более 15000 строк кода, и это только тех, которые относятся к ядру процессора, а ведь есть еще демонстрационные программы, которые заняли около 5000 строк. Однако мне вовсе не жаль потраченных усилий, поскольку я верю, что они принесут пользу. Я никогда не стремился к тому, чтобы предоставить вам трехмерный работающий процессор и сказать: "Пользуйтесь и наслаждайтесь!".

Вместо этого я шаг за шагом создаю на ваших глазах несколько трехмерных процессоров, надеясь продемонстрировать сопутствующий мыслительный процесс.

Мое восхищение той производительностью, которую демонстрирует 16-битовое ядро разрабатываемого процессора, настолько велико, что эта глава может оказаться последней из тех, в которой идет речь о 8-битовом режиме, поскольку в нем больше нет необходимости. Когда я приму такое решение, то сообщу об этом. А теперь приступим к освоению материала. Работая над этой главой, мы собираемся:

- обсудить новый процессор;
- обновить структуры с трехмерными данными;
- переписать загрузчики объектов;
- повторно рассмотреть растеризацию многоугольников;
- реализовать затенение по Гуро;
- рассмотреть основы теории дискретизации;
- реализовать аффинное отображение текстур;
- добавить освещение текстур;
- провести функциональный разбор листингов;
- рассмотреть стратегии оптимизации для 8- и 16-битовых режимов.

## Особенности нового трехмерного процессора

В конце предыдущей главы мы остановились на том, что наш процессор приобрел возможность выводить на экран заполненные объекты с применением полноцветной осветительной системы, поддерживающей источники общего освещения, бесконечно удаленные и точечные источники, а также световые пятна в модели освещения с плоским затенением. Кроме того, была начата разработка системы материалов и создано несколько загрузчиков файлов, а также вспомогательные программы, предназначенные для загрузки файлов в трехмерных форматах Studio .ASC и Caligari .COB. В настоящей главе мы рассмотрим вопросы затенения по Гуро и отображения текстур.

Функцию для отображения текстур или затенения по Гуро написать нетрудно, однако проблема в том, чтобы интегрировать эту функцию в процессор так, чтобы ею было удобно пользоваться. Любой может написать программу, имитирующую вращение перед камерой трехмерного освещенного куба с текстурой, а вот разработать обобщенную систему, обладающую возможностями моделировать камеру, источники освещения, работать с трехмерными моделями, текстурами, разрешением, битовой глубиной, форматами файлов и т.п. — это уже не так просто. Именно этим мы и займемся. Мне пришлось переписать весь процессор, чтобы добавить в него некоторые тонкие детали, модули освещения, загрузчики моделей объектов и другие модули. Кроме того, я внес определенную оптимизацию, например, добавил предварительное вычисление длин нормалей на этапе плоского освещения многоугольников, а также провел оптимизацию других видов в модулях освещения, перенес на предварительную стадию некоторые математические операции, что позволило ускорить работу программ.

Несмотря на то, что немало времени было потрачено на обсуждение концепции материалов и баз данных, в которые заносится информация о материалах, я все еще не готов интегрировать в процессор полнофункциональную систему, моделирующую материалы. Чтобы добавить возможность поддержки текстур, координат текстур, а также затенение по Гуро, придется изменить структуры данных и создать вторую версию многих из них

(вскоре мы обсудим, как это сделать). Однако при этом понадобится переписать все функции, по крайней мере, те из них, которые работают с трехмерными объектами.

Конечно же, во многих случаях это сведется к изменению прототипов и копированию кода. Однако в некоторых ситуациях придется учитывать наличие в структурах данных новых элементов. Более того, в системы освещения и обработки информации о цвете тоже будут внесены изменения, что повлияет на то, где будет храниться цвет многоугольника, полученный после его освещения.

Некоторые вопросы, от которых зависит производительность программ, действительно оказались безотлагательными. Назрела необходимость оптимизировать код, и откладывать это нельзя. Другими словами, в некоторых местах (например, при разработке модуля для отображения текстур) для повышения производительности пришлось прибегнуть к низкоуровневым трюкам. Тем не менее, обошлось без ассемблера, однако я здорово устал от использования SIMD-команд, позволяющих выполнять параллельную обработку данных, что может ускорить работу кода, как минимум, в четыре раза. Однако все это будет описано в главе, посвященной оптимизации.

Подводя итоги, заметим, что новая версия процессора поддерживает 8- и 16-битовые режимы, затенение по Гуро и освещение для одноцветных многоугольников, а также плоское затенение для многоугольников с текстурой, причем все это работает в полноцветном RGB-формате. Далее мы рассмотрим некоторые фрагменты кода, входящие в состав разрабатываемого нами процессора. Обещаю, что они не будут слишком длинными, потому что я не люблю нагромождений кода, за которыми не стоят теоретические пояснения. Итак, посмотрим, чему можно научиться при анализе кода нового процессора.

#### ВНИМАНИЕ

В процессе разработки второй версии процессора было сделано предположение о том, что в 16-битовой модели вычисления выполняются в формате 5.6.5. Такое предположение пришлось сделать по той причине, что разрешение этого вопроса "на ходу" отрицательно влияет на производительность. Можно было бы использовать условную компиляцию, сохранив скорость работы для обоих форматов, однако, честно говоря, среди известных мне видеокарт не удалось обнаружить таких, что работали бы с форматом 5.5.5.

## Обновление и разработка структур трехмерных данных

Прежде всего, приведем имена модулей, в которых содержатся новые библиотеки:

T3DLIB7.CPP — основные функции на языках C и C++;

T3DLIB7.H — заголовочный файл.

Для создания исполняемых файлов любой из программ, встречающихся в данной главе, их исходный код, как обычно, нужно скомпилировать и скомпоновать со всеми другими библиотечными модулями (от T3DLIB1.CPP до T3DLIB7.CPP). Как уже упоминалось во вступлении к этой главе, ее цель — добавить затенение по Гуро и аффинные отображения текстур (то есть линейные отображения, не принимающие во внимание перспективу). На первый взгляд этот материал кажется простым, однако в ходе более глубокого анализа оказывается, что многие изменения должны затрагивать структуры данных, чтобы добавить возможность поддержки новых особенностей освещения и наложения текстур. Конечно же, придется разработать и новые функции.

Приятно, что большинство функций можно было бы перенести из предыдущей версии процессора, изменив буквально несколько строк кода и, конечно же, прототип. Итак, ознакомимся с ключевыми элементами нового заголовочного файла T3DLIB7.H и убедимся, насколько важную роль они играют в разрабатываемом процессоре.

**СОВЕТ**

Все, кто привык работать с C++, возможно, не без злорадства скажут, что на C++ это сделать совсем несложно! Однако проблема все же существует. Это правда, что можно было бы просто создать производный класс и реализовать в нем необходимые изменения, но это привело бы к созданию нового класса, а я не хочу создавать иерархию классов. Я добиваюсь, чтобы в конечной программе была представлена обычная структура данных, с которой бы работал определенный набор функций. Таким образом, разрабатываемый процессор в этом смысле будет обладать наивысшей эффективностью.

**ВНИМАНИЕ**

Это предупреждение несколько выпадает из контекста. При ознакомлении с какой-либо из демонстрационных программ, работающих в 8-битовом режиме, иногда возникает задержка выполнения (до 1 минуты на слабых машинах). Это связано с вычислением таблиц преобразования. На этом этапе экран будет черным, но не стоит об этом беспокоиться.

## Новые директивы #define

Ниже приведены новые директивы #define, объявленные в новом библиотечном модуле T3DLIB7.CPP. Конечно же, все они содержатся в заголовочном файле T3DLIB7.H. Рассмотрим их раздел за разделом. Здесь имеется несколько констант, помогающих функциям, которые осуществляют отображение текстур, выводить треугольники.

```
// Эти макросы определяются для анализа треугольников
// в функции, выполняющей отображение текстуры.
#define TRI_TYPE_NONE 0
#define TRI_TYPE_FLAT_TOP 1
#define TRI_TYPE_FLAT_BOTTOM 2
#define TRI_TYPE_FLAT_MASK 3
#define TRI_TYPE_GENERAL 4
#define INTERP_LHS 0
#define INTERP_RHS 1
#define MAX_VERTICES_PER_POLY 6
```

Приведенные выше константы используются внутри системы определения состояния, входящей в состав функции для наложения текстуры, чтобы облегчить вывод на экран. Возможно, вам никогда не придется с ними сталкиваться, однако с этими константами связан один интересный прием. С их помощью мне удалось сделать так, чтобы функции производили визуализацию треугольников общего вида, а не разбивали их на два специальных треугольника: один с нижней горизонтальной стороной, а другой — с верхней.

Следующая группа инструкций #define определена для структуры POLY4DV2, которая вскоре будет рассмотрена подробнее.

```
// Эти директивы определены для многоугольников
// и поверхностей. Версия 2.

// Атрибуты многоугольников и элементов их поверхностей.
#define POLY4DV2_ATTR_2SIDED 0x0001
#define POLY4DV2_ATTR_TRANSPARENT 0x0002
#define POLY4DV2_ATTR_8BITCOLOR 0x0004
#define POLY4DV2_ATTR_RGB16 0x0008
#define POLY4DV2_ATTR_RGB24 0x0010

#define POLY4DV2_ATTR_SHADE_MODE_PURE 0x0020
#define POLY4DV2_ATTR_SHADE_MODE_CONSTANT 0x0020
// псевдоним
```

```

#define POLY4DV2_ATTR_SHADE_MODE_EMISSIVE 0x0020
    // псевдоним

«define POLY4DV2_ATTR_SHADE_MODE_FLAT 0x0040
#define POLY4DV2_ATTR_SHADE_MODE_GOURAUD 0x0080
#define POLY4DV2_ATTR_SHADE_MODE_PHONG 0x0100
#define POLY4DV2_ATTR_SHADE_MODE_FASTPHONG 0x0100
    // псевдоним
#define POLY4DV2_ATTR_SHADE_MODE_TEXTURE 0x0200

// Новая часть.
#define POLY4DV2_ATTR_ENABLE_MATERIAL 0x0800
    // Использование материала для освещения
#define POLY4DV2_ATTR_DISABLE_MATERIAL 0x1000
    // Для освещения используется только
    // основной цвет (эмулируется версия 1.0)

// Состояние многоугольников и элементов их поверхностей
#define POLY4DV2_STATE_NULL 0x0000
#define POLY4DV2_STATE_ACTIVE 0x0001
#define POLY4DV2_STATE_CLIPPED 0x0002
#define POLY4DV2_STATE_BACKFACE 0x0004
#define POLY4DV2_STATE_LIT 0x0008

```

Если вы хоть немного экспериментировали с процессором, то поняли, что почти все директивы `ftdefine` с именем `POLY4DV2_*` идентичны соответствующим директивам с именем `POLY4DV1_*`, объявленным в предыдущей версии. Добавлено лишь несколько новых директив, которые выделены жирным шрифтом. Как и в предыдущей версии, новые директивы `ftdefine` помогут найти правильное место для некоторых структур, в которых впоследствии будут заноситься данные о материалах, а также пометить освещенные многоугольники. Это нужно сделать, поскольку есть несколько вариантов построения последовательности, сформированной различными этапами, на которые разбивается моделирование освещения объектов (по крайней мере, их будет несколько к концу главы). Если многоугольник уже освещен, то странно было бы освещать его снова!

Далее идут директивы `#define`, соответствующие различным флагам, которые относятся к вершинам многоугольника. Эти флаги передаются при загрузке объектов функциями-загрузчиками. Они позволяют изменять такие свойства объекта, как модель затенения, а также помогают выполнить более простым путем такие действия, как инвертирование координат текстур и другие.

```

// (Новый блок.) Эти директивы используются для простых
// форматов моделей и помогают лучше управлять освещением
#define VERTEX_FLAGS_OVERRIDE_MASK 0xf000 // Маскировка
    // битов для извлечения

«define VERTEX_FLAGS_OVERRIDE_CONSTANT 0x1000
«define VERTEX_FLAGS_OVERRIDE_EMISSIVE 0x1000 // Псевдоним
#define VERTEX_FLAGS_OVERRIDE_PURE 0x1000
#define VERTEX_FLAGS_OVERRIDE_FLAT 0x2000
#define VERTEX_FLAGS_OVERRIDE_GOURAUD 0x4000
#define VERTEX_FLAGS_OVERRIDE_TEXTURE 0x8000

#define VERTEX_FLAGS_INVERT_TEXTURE_U 0x0080
    // Инвертирование координаты текстуры и

```

```

tdefine VERTEX_FLAGS_INVERT_TEXTURE_V 0x0100
// Инвертирование координаты текстуры v
#define VERTEX_FLAGS_INVERT_SWAP_UV 0x0800
// Перестановка координат текстуры u и v

```

В процессе обсуждения функций, осуществляющих загрузку объектов, мы разберемся со всем, что касается перезаписи информации о вершинах. Давайте продолжим и перейдем к следующему блоку директив `#define`, связанному со второй версией структуры `OBJECT4DV2`, предназначенной для хранения параметров объекта. Как и в предыдущей структуре, большинство директив дублируется. Немного изменены лишь имена идентификаторов, которые встречаются в этих директивах; теперь они заданы в виде `OBJECT4DV2_*`. Однако есть и несколько новых интересных директив. В приведенном ниже фрагменте кода они выделены жирным шрифтом.

```

// Директивы определены для объектов версии 2.
// Память для объектов выделяется динамически,
// однако ее максимальный объем ограничен
#define OBJECT4DV2_MAX_VERTICES 1024
#define OBJECT4DV2_MAX_POLYS 2048

// Состояния объектов
#define OBJECT4DV2_STATE_NULL 0x0000
#define OBJECT4DV2_STATE_ACTIVE 0x0001
#define OBJECT4DV2_STATE_VISIBLE 0x0002
#define OBJECT4DV2_STATE_CULLED 0x0004

// Новые директивы
#define OBJECT4DV2_ATTR_SINGLE_FRAME 0x0001
// Однокаркасный объект (эмулируется версия 1.0)
#define OBJECT4DV2_ATTR_MULTI_FRAME 0x0002
// Многокаркасный объект для поддержки формата .md2 и др.
#define OBJECT4DV2_ATTR_TEXTURES 0x0004
// Флаги, определяющие, содержит ли объект многоугольники
// с текстурой

```

Представляют интерес только три последние директивы `tdefine`. Объявление идентификаторов `OBJECT4DV2_ATTR_SINGLE_FRAME` и `OBJECT4DV2_ATTR_MULTI_FRAME` помогает поддерживать объекты, в которых задано несколько каркасов. Не забывайте о том, что мы стремимся к использованию формата, способного поддерживать анимацию, например *Quake II*.MD2. Для этого нужно каким-то образом пометить объект, чтобы указать, что в списках его вершин содержится несколько каркасов. После того как выбран "текущий" каркас, должны использоваться только вершины этого каркаса, хотя эти вершины соответствуют одним и тем же многоугольникам, цветам и другим подобным параметрам других каркасов. Более подробно этот вопрос будет рассмотрен позже. Наконец, атрибут `OBJECT4DV2_ATTR_TEXTURES` представляет флаги, которые указывают, есть ли текстуры где-нибудь в объекте, что помогает выполнять определенную оптимизацию.

Далее задается максимально допустимое количество многоугольников в списке визуализации. Это значение задается произвольно, поэтому его можно изменять. Однако я сомневаюсь, что мы сможем визуализировать в одном кадре больше 10000 многоугольников при скорости анимации 15-30fps, поэтому думаю, что 32768 многоугольников — это более чем достаточно.

```

// Определение для версии 2.0 списка визуализации
#define RENDERLIST4DV2_MAX_POLYS 32768

```

Довольно интересен набор директив `ttdefine`, который представлен ниже (почему это так — становится понятно из комментариев). Далее в определении вершин будет содержаться намного больше информации, чем раньше. В таких определениях будет содержаться не только положение самой вершины, но и вектор нормали, координаты текстуры и флаги атрибутов.

```
// Определения вершин, содержащие вспомогательную информацию
// для преобразования объектов и их освещения. Это помогает
// определить, имеет ли данная вершина корректную нормаль,
// которую необходимо повернуть, или координаты текстуры,
// которые нужно отсечь и т.д. Это поможет уменьшить
// загрузку при моделировании освещения и выводе
// изображения, поскольку мы точно знаем, с вершиной какого
// вида мы имеем дело. По количеству содержащейся информации
// это определение напоминает такой гибкий формат
// определения вершин, как формат direct3d, который может
// быть задан в одной из таких форм:
// координаты точки;
// координаты точки + нормаль;
// координаты точки + нормаль + координаты текстуры.
#define VERTEX4DTV1_ATTR_NULL 0x0000 // Пустая вершина
typedef struct VERTEX4DTV1_ATTR_POINT 0x0001
#define VERTEX4DTV1_ATTR_NORMAL 0x0002
#define VERTEX4DTV1_ATTR_TEXTURE 0x0004
```

Я возлагаю большие надежды на новую структуру данных `VERTEX4DTV1`, поскольку она поможет сэкономить много времени, однако отложим ее определение до тех пор, пока мы не перейдем к рассмотрению структур данных.

Вот и все, что нужно было сказать по поводу добавленных директив `#define`. Я же говорил, что здесь не так много нового. Важно последовательно повышать сложность процессора. Таким образом можно многому научиться и в то же время получить богатый опыт, причем отрицательный опыт тоже полезен.

## Добавление математических структур

Вы уже ознакомились с недавно созданной математической библиотекой? Там есть на что посмотреть! С каждой новой главой нам нужно было добавлять очень немного математических функциональных возможностей — лишь некоторые новые структуры данных с более понятными именами. В данной главе — это типы векторов с целочисленными координатами. Приведем их определения.

```
// Двумерный вектор с целочисленными координатами
// Точка без координаты w
typedef struct VECTOR2DI_TYP
{
    union
    {
        int M[2]; // Массив
        // Явные имена.
        struct
        {
            int x, y;
        }; // struct
    }
};
```

```

}; // union
} VECTOR2DI, POINT2DI, *VECTOR2DI_PTR, *POINT2DI_PTR;

// Трехмерный вектор с целочисленными координатами
// Точка без координаты w
typedef struct VECTOR3DI_TYP
{
    union
    {
        int M[3]; // Массив
        // Явные имена
        struct
        {
            int x,y,z;
        }; // struct
    }; // union
} VECTOR3DI, POINT3DI, *VECTOR3DI_PTR, *POINT3DI_PTR;

// Четырехмерный вектор с целочисленными координатами
// Точка без координаты w
typedef struct VECTOR4DI_TYP
{
    union
    {
        int M[4]; // Массив
        // Явные имена.
        struct
        {
            int x,y,z,w;
        }; // struct
    }; // union
} VECTOR4DI, POINT4DI, *VECTOR4DI_PTR, *POINT4DI_PTR;

```

По сути, выше приведены определения двумерных, трехмерных и четырехмерных векторов с целочисленными координатами и ничего более. Эти определения помогут при реализации отображения текстур или сохранении целочисленных данных в векторной форме, которые не хочется преобразовывать в числа с плавающей точкой.

## Вспомогательные макросы

Ниже приведено определение полезной функции, позволяющей сравнивать числа с плавающей точкой.

```

// Сравнение чисел с плавающей точкой
#define FCMP(a,b) ((fabs(a-b) < EPSILON_E3) ? 1 : 0)

```

Не знаю, известно ли вам об этом, но при работе с числами с плавающей точкой лучше не пользоваться следующим кодом.

```

float f1, f2;
// Переменным f1 и f2 присваиваются некоторые значения
if (f1==f2)
{
    // Выполнение необходимых действий
}

```

Проблема в условном операторе if. Даже если значения переменных f1 и f2 на самом деле идентичны, их **машинное** представление может различаться в результате округления и недостаточной точности. Таким образом, сравнение с **помощью** приведенного выше кода может не дать правильного результата. Поэтому лучше вычислить модуль разности этих чисел и сравнить его с некоторой малой величиной, например, 0.001 или 0.0001. На практике оба эти числа можно считать равными 0.0, поскольку точность чисел с **плавающей** точкой в лучшем случае сохраняется лишь в 4-5 десятичном знаке. Вот почему нам понадобился макрос FCMP().

Приведенные ниже встраиваемые макросы — это просто копии функций, выполняющих копирование одних новых объектов VERTEX4DTV1 в другие. Технически операторы копирования для них не нужны, однако пока все не будет оптимизировано, я предпочитаю давать подробные пояснения.

```
inline void VERTEX4DTV1_COPY(VERTEX4DTV1_PTR vdst,
                             VERTEX4DTV1_PTR vsrc)
{ *vdst = *vsrc; }
```

```
inline void VERTEX4DTV1_INIT(VERTEX4DTV1_PTR vdst
                             VERTEX4DTV1_PTR vsrc)
{ *vdst = *vsrc; }
```

Последняя встраиваемая функция представляет собой несколько более быструю версию функции вычисления длин векторов. Единственное различие состоит в том, что в старой функции VECTOR4D\_Length\_Fast() вызывалась функция, производящая общее **вычисление** длин в трехмерном пространстве, и мне этот лишний вызов не нравился. Поэтому пришлось разработать немного улучшенный вариант функции.

```
inline float VECTOR4D_Length_Fast2(VECTOR4D_PTR va)
{
    // Эта функция вычисляет расстояние от начала координат
    // до точки x,y,z

    int temp; // Используется для обмена
    int x,y,z; // Используются в алгоритме

    // Проверим, все ли величины положительны
    x = fabs(va->x) * 1024;
    y = fabs(va->y) * 1024;
    z = fabs(va->z) * 1024;

    // Сортировка величин
    if (y < x) SWAP(x,y,temp)
        if (z < y) SWAP(y,z,temp)
            if (y < x) SWAP(x,y,temp)
                intdist = (z + 11*(y >> 5) + (x >> 2));
    // Вычисление расстояния с ошибкой, составляющей 8%
    return((float)(dist >> 10));
} // VECTOR4D_Length_Fast2
```

## Новые возможности представления трехмерных каркасов

Обновление структур данных, предназначенных для хранения информации о трехмерных объектах, далось нелегко. Бессмысленно сразу заглядывать на последнюю стра-

ницу (если вы понимаете, о чем я говорю), поэтому я предпочитаю плавно переходить от версии к версии, делая лишь те добавления, которые необходимы для выполнения поставленной задачи.

Давайте вспомним материал предыдущей главы, в которой была реализована полноцветная осветительная система, поддерживающая как излучаемый свет, так и плоское затенение на уровне отдельных многоугольников. Форматы **PLG/PLX** и **.COB** позволяют создавать модели, в **которых** можно задавать **свойства** каждого многоугольника (а формат **.ASC** поддерживает только вершины, многоугольники и цвета). В формате **PLG/PLX** можно вручную задавать вид затенения многоугольников. Это делается с помощью соответствующих флагов.

```
// Определения для улучшенного формата PLG -> PLX
// Дескриптор поверхности все еще 16-битовый
// Теперь его формат имеет такой вид:
// d15      d0
// CSSD | RRRR | GGGG | BBBB
// C - это флаг цвета: в формате RGB или индексированный;
// SS - это два бита, в которых задается режим затенения;
// D - это флаг двусторонней поверхности;
// RRRR, GGGG, BBBB - это биты, содержащие интенсивности
//           красного, зеленого и синего каналов
//           для режима RGB или
// GGGGBBBB - это 8-битовый индекс цвета для
//           8-битового режима

// Битовые маски, упрощающие тестирование
#define PLX_RGB_MASK      0x8000 // Маска для извлечения
// цвета в формате RGB
// или индексированном
// формате
#define PLX_SHADE_MODE_MASK 0x6000 // Маска для извлечения
// режима затенения
#define PLX_2SIDED_MASK    0x1000 // Маска для двусторонних
// поверхностей
#define PLX_COLOR_MASK     0x0fff // xxxrrrrgggbbbb, по 4
// бита для каждого канала в формате RGB;
// xxxxxxxxiiiiii, 8-битовый индекс

// Это флаги сравнения, применяющиеся после преобразования
// цветового режима многоугольника
#define PLX_COLOR_MODE_RGB_FLAG  0x8000
// Цвет многоугольника задается в формате RGB
#define PLX_COLOR_MODE_INDEXED_FLAG 0x0000
// Цвет многоугольника задается в виде 8-битового индекса

// Флаг двусторонней поверхности
#define PLX_2SIDED_FLAG    0x1000 // Двусторонний
// многоугольник
#define PLX_1SIDED_FLAG    0x0000 // Односторонний
// многоугольник

// Режим затенения многоугольника
#define PLX_SHADE_MODE_PURE_FLAG  0x0000 // Одноцветный
```

```

// многоугольник
tfdefinePU_SHADE_MODEj;ONSTANT_FLAG 0x0000 // Псевдоним
ttdefine PLX_SHADE_MODE_FLAT_FLAG 0x2000 // Многоугольник
// с плоским затенением
#define PLX_SHADE_MODE_GOURAUD_FLAG 0x4000 // Многоугольник
// с затенением по Гуро
#define PLX_SHADE_MODE_PHONG_FLAG 0x6000 // Многоугольник
// с затенением по Фонгу
#define PLX_SHADE_MODE_FASTPHONG_FLAG 0x6000 // Многоугольник
// с затенением по Фонгу

```

Для формата .COB было решено, что поле color используется для определения цветных многоугольников и многоугольников с текстурой, а поле reflectance — для определения вида освещения, основанного на соглашениях, описанных в комментариях к загрузчику файлов в формате .COB.

```

// Теперь нужно задать модель затенения. Это не так просто;
// для этого нужно найти строки "Shader class: color", а
// потом - строку, которая имеет вид: "Shader name: "xxxxxx"
// (xxxxxx)", вместо последовательности символов xxxxxx для
// одноцветных многоугольников может присутствовать надпись
// "plain color" или "plain", а для текстур - надпись
// "texture map" или "caligari texture". Далее осуществляем
// поиск строки "Shader class: reflectance", в которой
// закодирован вид затенения. Потом снова ищем строку
// "Shader name: "xxxxxx" (xxxxxx)", и на основании ее
// значения выбираем атрибут в системе затенения. Это
// делается следующим образом:
// "constant" -> MATV1_ATTR_SHADE_MODE_CONSTANT;
// "matte" -> MATV1_ATTR_SHADE_MODE_FLAT;
// "plastic" -> MATV1_ATTR_SHADE_MODE_GOURAUD

```

Если вы помните, у нас была небольшая проблема, связанная с моделированием освещения. Мы не продумали достаточно хорошо структуры данных, предназначенные для хранения параметров многоугольников. В них негде хранить цвет многоугольника, полученный после его освещения. Другими словами, цвет многоугольника хранился в 8- или 16-битовом формате в поле color. Затем это значение использовалось для вычисления цвета освещенного многоугольника. Далее, когда освещение выполнялось в списке визуализации, не было ничего страшного в удалении или перезаписи информации о цвете многоугольника, поскольку он был на финишной прямой к выводу на экран. Проблема возникает тогда, когда моделирующая освещение функция вызывается для освещения исходного объекта, не разобранного на отдельные элементы и не помещенного в список визуализации.

Трудность обусловлена тем, что нельзя терять исходный цвет многоугольника, перезаписывая его цветом, полученным в результате работы симулятора освещения. Поэтому наша стратегия заключается в том, чтобы скопировать полученный в результате освещения цвет в старшие 16 битов переменной, хранящей цвет многоугольника. Описанная схема проиллюстрирована на рис. 9.1.

В предыдущей версии игрового процессора этот трюк работал прекрасно. Однако этот метод уже не годится, если многоугольники освещаются с применением затенения по Гуро. Таким образом, одним из основных добавлений, внесенных в структуры данных многоугольников, станет поддержка полей, предназначенных для хранения конечного цвета, чтобы не исказить информацию об исходном цвете,

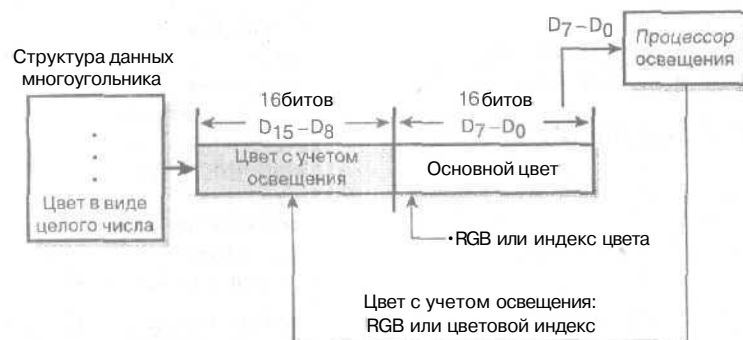


Рис. 9.1. Сохранение цвета освещенного многоугольника в старших 16 битах

Следующий вопрос — это сам алгоритм затенения по Гуро. Обратите внимание на рис. 9.2, благодаря которому видно, что нам понадобится нормаль к каждой вершине (это основной момент в алгоритме Гуро: сначала находится интенсивность освещения в каждой вершине, а затем выполняется интерполяция). Поэтому еще один вопрос, который следует продумать, — как добавить конечный цвет каждой вершины. При этом еще раз придется позаботиться о месте для хранения цвета вершин каждого многоугольника с учетом освещения.

Наконец, последнее, что нужно добавить, — поддержка анимации каркасов, подобная той, которая существует в формате *Quake II*.MD2. Эта концепция проиллюстрирована на рис. 9.3. По сути, анимация каркасов аналогична стандартной анимации битовых изображений, которая представляет собой набор кадров, содержащих последовательные тонированные изображения движущегося объекта. Анимация каркасов работает точно так же. Например, пусть у нас есть куб, в состав которого входят 8 вершин и 12 многоугольников. Для каждого многоугольника заданы цвет и модель затенения, причем каждый многоугольник определен с помощью трех вершин, содержащихся в главном списке вершин, в котором всего находится 8 вершин. Теперь нужно каким-то образом добавить еще один каркас. Для этого создаются 8 новых вершин (в том же порядке, что и предыдущие), несколько смещенных по отношению к предыдущим; теперь эти вершины будут заменять предыдущие 8 вершин. Многоугольники по-прежнему определены. Для них заданы и цвета, и текстуры, но теперь объект немного сместился!

В формате *Quake II*.MD2 используется точно такой же метод. Единственное правило, которое нужно соблюдать, — это сохранение количества вершин и многоугольников, из которых состоит движущийся каркас. Можно менять положения вершин, но не их взаимосвязь с многоугольниками; например, если многоугольник 22 состоит из вершин (19,90,200), в дальнейшем он всегда должен состоять именно из этих вершин.

В любом случае мы не станем в этой главе разрабатывать программу для считывания файлов в формате .MD2. Однако поддержка анимации для файлов в этом формате будет добавлена очень скоро, и поскольку эта функциональная особенность играет большую роль, я добавил ее в структуру *OBJECT4DV2*, что следует принять во внимание в большинстве заново переписываемых функций.

С учетом вышесказанного последовательно рассмотрим структуры данных, предназначенные для хранения информации о вершинах и многоугольниках.

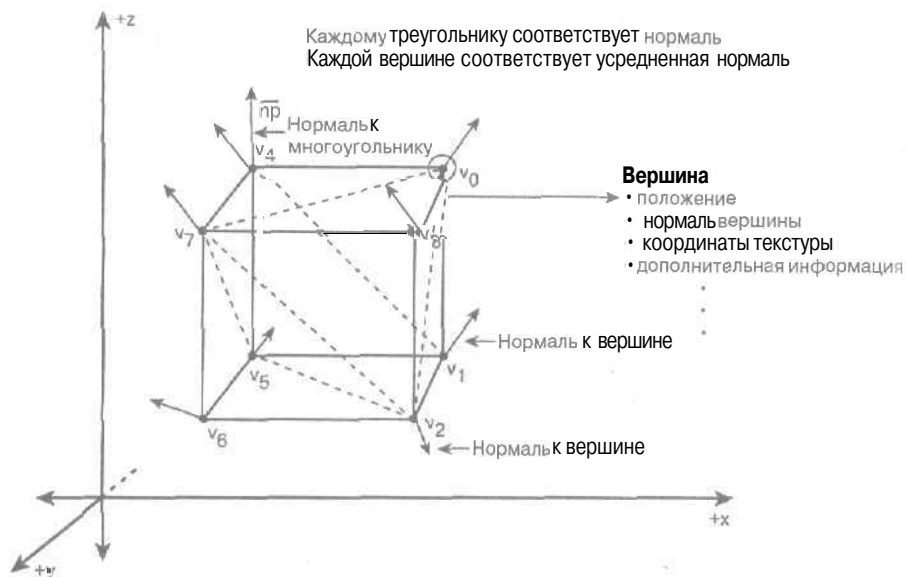


Рис. 9.2. Для моделирования затенения по Гуро необходимо вычислить нормали в вершинах



Рис. 9.3. Объекты, поддерживающие несколько каркасов

### Первая реальная вершина

До настоящего времени тип, который соответствует **вершине**, не определен у нас в том виде, в котором это необходимо для полнофункциональной реализации трехмерной графики. Этот тип определен просто как точка в трехмерном пространстве. Настало время создать более полные структуры данных, **отвечающие** новым требованиям. Кроме того, с помощью новых полей признака (определенных ранее с помощью директив `#define VERTEX4DTV1_ATTR_*`) я попытался добавить новые возможности управления, что позволит придать гибкость формату вершин. Таким образом можно создать новые типы вершин, содержащих различную информацию; при этом структура и компоновка должны всегда **оставаться** одинаковыми. Ниже приведена новая структура данных **VERTEX4DTV1**.

НА ЗАМЕТКУ

Те, кто использует в своих программах **Direct3D**, понимают преимущества и важность гибких форматов вершин. До их появления программисты вынуждены были пользоваться **предопределенными** форматами вершин, что усложняло процесс программирования.

// Четырехмерные однородные вершины с двумя координатами  
// текстуры и нормалью к вершине; нормаль можно трактовать

```

// как вектор или точку
typedef struct VERTEX4DTV1_TYP
{
    union
    {
        float M[12]; // Массив

        // Явные имена
        struct
        {
            float x,y,z,w; // Положение точки
            float nx,ny,nz,nw; // Нормаль (вектор или точка)
            float u0,v0; // Координаты текстуры

            float i; // Интенсивность света в вершине
                        // после освещения
            int attr; // Атрибуты или дополнительные
                        // координаты текстуры
        }; // struct

        // Типы высокого уровня
        struct
        {
            POINT4D v; // Вершина
            VECTOR4D n; // Нормаль
            POINT2D t; // Координаты текстуры
        };
    }; // union
} VERTEX4DTV1, *VERTEX4DTV1_PTR;

```

Данная структура представляет собой объединение нескольких типов, определенных для упрощения доступа к данным, в зависимости от того, как будет разрабатываться алгоритм. Например, если описывающие вершину данные представлены в виде потока (как это сделано в технологии Direct3D), может оказаться, что доступ к ним удобнее получать как к элементам массива. Вот зачем в объединении понадобилось задавать массив `M[]`. С другой стороны, кто-то может предпочесть пользоваться именами, заданными на более высоком уровне, поэтому для каждой части вершины определены явные имена:

`x, y, z, w` — координаты вершины;

`nx, ny, nz, nw` — нормаль к вершине (подробнее об этом будет сказано позже);

`u0, v0` — координаты двумерной текстуры;

`i` — интенсивность или цвет вершины;

`attr` — дескрипторы вершины, указывающие ее тип и содержимое.

Кроме того, в других безымянных объединениях содержатся другие структурированные имена: `v`, `n` и `t`. В них хранятся данные, описывающие вершину, нормаль и координаты текстуры, соответственно. Для атрибутов и интенсивности высокоуровневые имена не заданы, поскольку на самом деле это не компоненты. Представленное выше объединение используется автоматически, поскольку у него нет имени. Например, чтобы получить доступ к адресу трехмерной точки, следует использовать следующий код,

```

VERTEX4DTV1 p1;
&p1.v

```

Доступ к абсциссе трехмерной точки можно получить с помощью такой инструкции и `&p1.x`

или с помощью объединения `v —`

`&p1.v.x`

Здорово, правда?

Такой подход удобен тем, что с вершиной, нормалью или координатами текстуры можно оперировать как со структурами, а можно перейти на уровень компонент. Наконец, теперь у нас появилась возможность задавать в процессе создания вершины нормаль (она важна при моделировании **освещения**) и координаты текстуры, однако это делается только в случае необходимости. Это достигается путем использования поля `attr`, входящего в состав типа вершины. Например, если нужно, чтобы этот тип поддерживал положение точки в трехмерном пространстве и нормаль, но не координаты текстуры, следует воспользоваться **следующими** настройками.

```
VERTEX4DTV1 p1;
p1.attr = VERTEX4DTV1_ATTR_POINT | VERTEX4DTV1_ATTR_NORMAL;
```

Конечно, это поле не гарантирует защиту от доступа к точке, нормали или координатам текстуры. Оно просто используется как сообщение, позволяющее программисту и созданным им функциям получить информацию о том, что правильно, а что — нет.

### Новые внутренние структуры POLY4DV2

Еще одно значительное изменение в нашем игровом процессоре касается структур данных, в которых хранится информация о многоугольниках. Напомним, что у нас есть две такие структуры: одна из них относится к внешним спискам вершин и сама является частью структуры (с данными об объекте), а другая содержит всю необходимую информацию для списка визуализации. Взаимоотношения между различными переменными новой структуры POLY4DV2 и элементами, участвующими в формировании изображения, проиллюстрированы на рис. 9.4. Ниже приводится само определение структуры.

```
// Версия 2.0 многоугольника, заданного на основе внешнего
// списка вершин
typedef struct POLY4DV2_TYP
{
    int state;           // Информация о состоянии
    int attr;           // Физические атрибуты многоугольника
    int color;          // Цвет многоугольника
    int lit_color[3];    // Здесь хранится цвет после освещения;
                        // в случае плоского затенения задаются нули
                        // 0,1,2 - индексы цветов вершин после их освещения

    BITMAP_IMAGE_PTR texture; // Указатель на информацию о
                        // текстуре для ее простого отображения

    int mati;           // Индекс (-1) обозначает отсутствие
                        // материала (новая особенность)

    VERTEX4DTV1_PTR vtlist; // Список вершин
    POINT2D_PTR tlist;      // Список текстур
    int vert[3];           // Индексы списка вершин
    int text[3];           // Индексы в списке координат вершин
    float nlength;         // Длина нормали
} POLY4DV2, *POLY4DV2_PTR;
```

```
typedef struct POLY4DV2_TYP
```

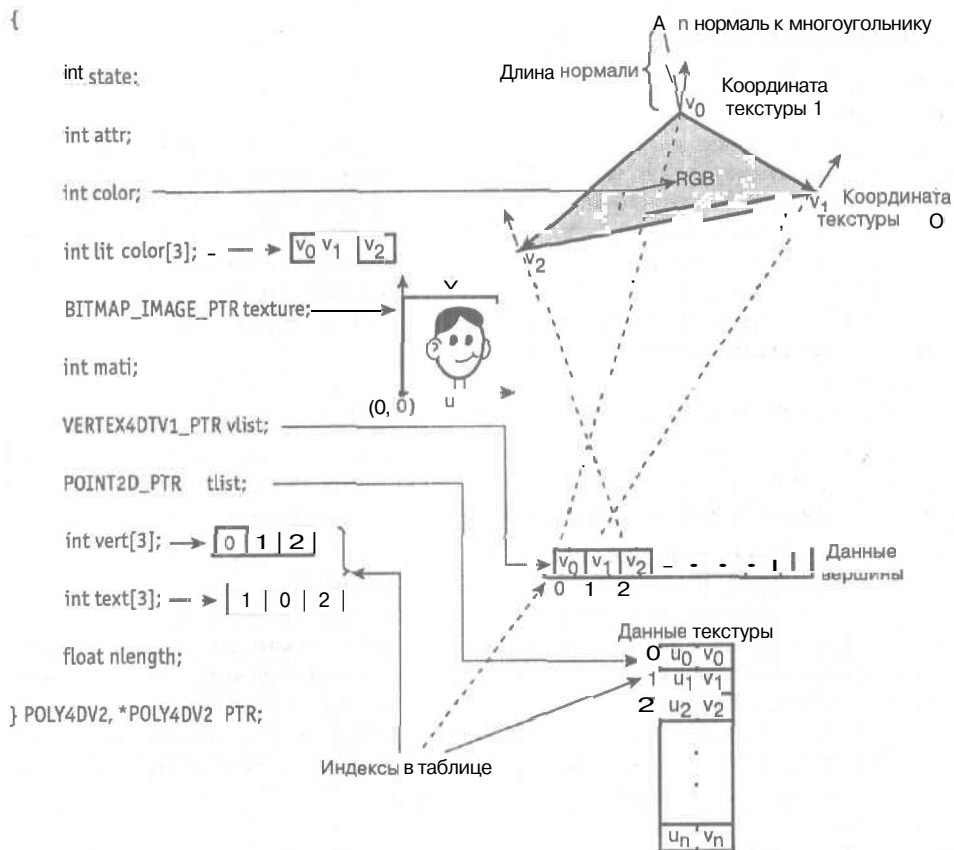


Рис. 9.4. Графическое представление новой структуры POLY4DV2

Полужирным шрифтом отмечены все новые и измененные поля структуры. Как видим, их довольно много! Обсудим назначение каждого из этих полей.

- **lit\_color[3]**. В новую осветительную систему понадобилось добавить поддержку не только одного цвета, заданного для всего **треугольника**, но и **еще** трех цветов, заданных для его вершин. Эти цвета используются, когда необходимо выполнить затенение по **Гуро**. Указанная особенность поддерживается с помощью поля **lit\_colors[3]**. Если вычисление интенсивности освещения выполняется для многоугольника с плоским затенением, результат не записывается в верхние 16 битов поля **color**, а заносится в элемент массива **lit\_colors[0]**. Аналогично, при имитировании затенения по Гуро конечные цвета каждой вершины соответственно заносятся в элементы массива **lit\_colors[0,1,2]**.
- **texture**. Во избежание излишних усложнений, я по-прежнему пытаюсь обходиться без полноценной поддержки материалов. Однако все же нужно выделить место для хранения текстурного отображения каждого многоугольника, обладающего текстурой. Данное поле представляет собой указатель на отображение **текстуры**, если оно существует для рассматриваемого **многоугольника**.

- **mati.** Данное поле является индексом материала в базе данных, в которой хранятся сведения о материалах. Оно предназначено для запланированной на будущее поддержки материалов. Я до конца не уверен, понадобится ли нам это поле, но решил добавить его на данном этапе, чтобы в дальнейшем не возникло никаких проблем.
- **tlist.** Это поле предназначено для добавленной поддержки отображения текстуры. Как видно из рис. 9.4, при наложении текстуры на многоугольник необходимо указать координаты текстуры. Поэтому в новой структуре **OBJECT4DV2** задан массив с координатами текстуры, проиндексированный переменной **tlist** аналогично тому, как индексы массива вершин хранятся в переменной **vlist**. Эти координаты текстуры впоследствии используются для отображения.
- **text[3].** Фактически это индексы текстуры в списке координат текстур.
- **nlength.** В этом поле содержится длина вектора нормали, проведенного к многоугольнику, что способствует ускорению вычислений, связанных с освещением. Эта оптимизация не так уж важна, однако она не давала мне покоя. Если посмотреть на математические выражения, которые участвуют в осветительном процессоре, то можно увидеть, что нормаль к многоугольнику находится путем вычисления векторного произведения  $(\vec{v}_0 \vec{v}_1 \times \vec{v}_0 \vec{v}_2)$ . Впоследствии понадобится нормировать найденный в результате вектор или хотя бы вычислить его длину.

Меня не так беспокоит вычисление векторного произведения, как вычисление длины. Дело не столько в том, что при этом получается приближенное значение, сколько в том, что нет необходимости каждый раз вычислять длину вектора нормали после вращения или переноса многоугольника. Если для вычисления длины все время используется одна и та же пара векторов (а именно так и происходит), то в процессе преобразования многоугольника, не меняющего его форму, длина нормали не меняется. Таким образом, можно на предварительном этапе вычислить длины всех нормалей, а затем использовать их в дальнейших расчетах. В дополнение к рационализации вычислений мы получим вторичный эффект, состоящий в повышении точности, поскольку определение длин нормалей выполняется во время загрузки объекта. Это избавляет от необходимости пользоваться более быстрой, но приближенной функцией, предназначенной для работы в реальном времени. Короче говоря, с помощью описанной оптимизации экономится множество процессорного времени.

#### СОВЕТ

Может возникнуть вопрос: а почему бы таким же образом не вычислить сам вектор нормали? Это тоже можно было бы сделать — вычислить вектор нормали, а затем нормировать его. Однако впоследствии пришлось бы производить вращение при преобразованиях каркаса, а это лишняя работа. При вращении вектора с помощью матрицы поворота 4x4 приходится выполнять столько же операций, как и при вычислении векторного произведения, а то и больше. Поэтому, хотя эта идея сначала и кажется привлекательной, предварительное вычисление вектора нормали себя не оправдывает.

### Новые внешние самодостаточные структуры POLYF4DV2

Обновленная самодостаточная структура данных POLYF4DV2, содержащая описание многоугольника, предназначена для внесения в список визуализации окончательной информации о многоугольнике. Рассмотрим ее определение.

```
// Самодостаточная структура с описанием многоугольника
// для составления версии 2 списка визуализации
typedef struct POLYF4DV2_TYP
```

```

i
int state;           // Информация о состоянии
int attr;            // Физические атрибуты многоугольника
int color;           // Цвет многоугольника
int lit_color[3];    // Здесь хранится цвет после
                    // освещения; в случае плоского
                    // затенения задаются нули

BITMAP_IMAGE_PTR texture; // Указатель на информацию о
                    // текстуре для ее простого
                    // отображения

int mati;            // Индекс (-1) обозначает отсутствие
                    // материала (новая особенность)

float nlength;       // Длина вектора нормали, если он не
                    // нормирован (новая особенность)

VECTOR4D normal;     // Общая нормаль к многоугольнику
                    // (новая особенность)

float avg_z;         // Усредненная координата z вершин;
                    // используется для простой
                    // сортировки многоугольников (новая
                    // особенность)

VERTEX4DTV1 vlist[3]; // Вершины текущего треугольника
VERTEX4DTV1 tvlist[3]; // Вершины после преобразования,
                    // если оно необходимо

POLYF4DV2_TYP *next; // Указатель на следующий
                    // многоугольник в списке
POLYF4DV2_TYP *prev; // Указатель на предыдущий
                    // многоугольник в списке

} POLYF4DV2, *POLYF4DV2_PTR;

```

Структуры `POLYF4DV2` и `POLY4DV2` почти идентичны. Единственное различие заключается в том, что входящее в состав структуры `POLYF4DV2` хранилище информации о вершинах является внутренним, а не внешним, т.е. не заданным по ссылке. Кроме того, в самой структуре `VERTEX4DTB1`, представляющей каждую вершину, теперь также содержатся координаты текстуры, поэтому вся необходимая информация о каждой вершине находится в массивах `vlist[]` и `tvlist[]`. Все поля структур `POLYF4DV2` и `POLY4DV2` с одинаковыми именами имеют одно и то же значение.

## Обновление структур объектов и списка визуализации

А теперь самое главное! При разработке исходного варианта игрового процессора было решено помещать объект в общую структуру, содержащую информацию как о вершинах, так и о многоугольниках, из которых состоит этот объект. Для объектов, передвигающихся по игровому полю, это вроде бы работало замечательно. Однако подойдет ли такой способ для других высокоуровневых данных, таких как уровни ифы и т.п.? Ответ в основном утвердительный. Фактически уровень можно скомпоновать путем объединения в рамках одного списка объектов, представляющих этаж помещения или комнату, поэтому можно сказать, что мы на верном пути. Конечно же, когда дело дойдет до раз-

мещения объектов в пространстве, станет понятно, какие необходимо сделать преобразования и изменения для поддержки **концепции**, представляющей геометрию уровня. Однако пока что определенные ранее структуры зарекомендовали себя как хорошие контейнеры, поэтому мы будем продолжать работу с ними.

Новая версия структуры **OBJECT4DV2**, содержащей информацию об объекте, почти аналогична предыдущей. Однако она, конечно же, поддерживает координаты текстуры, новые атрибуты и **общую** текстуру (к рассмотрению которой мы скоро перейдем). Новым важным моментом структуры **OBJECT4DV2** является поддержка **кадров**, с помощью которой представляется движение каркаса. Еще не написана ни одна программа, использующая эту особенность (в загрузчике файлов в формате **.MD2** можно было бы реализовать этот трюк), однако инфраструктура для поддержки анимации все же **разрабатывается**. Еще раз посмотрите на рис. 9.3, на котором схематически изображена новая структура **OBJECT4DV2**, и обратите внимание, как в ней организовано хранение информации о каждом каркасе. Ниже приведена сама структура. Обратите внимание на ее поля, выделенные полужирным шрифтом.

```
// Версия 2.0 структуры объекта, основанная на списке
// вершин и списке многоугольников. Новый объект обладает
// намного большей гибкостью и поддерживает покадровую
// анимацию, т.е. этот объект способен содержать сотни
// каркасов. При этом каркас должен состоять из одного и
// того же количества многоугольников и иметь одну и ту же
// геометрию. Положение вершины может изменяться подобно
// тому, как это делается в формате Quake II .md2
typedef struct OBJECT4DV2_TYP
{
    int id;                // Численный идентификатор объекта
    char name[64];         // ASCII-имя объекта (на всякий случай)
    int state;              // Состояние объекта
    int attr;              // Атрибуты объекта
    int mati;              // Если индекс материала равен -1, это
                          // означает, что материал отсутствует
                          // (новая особенность)
    float *avg_radius;     // [OBJECT4DV2_MAX_FRAMES]; // средний
                          // радиус объекта; используется для
                          // определения столкновений
    float *max_radius;     // [OBJECT4DV2_MAX_FRAMES];
                          // Максимальный радиус объекта

    POINT4D world_pos;     // Положение объекта в глобальной
                          // системе координат

    VECTOR4D dir;          // Углы вращения объекта в локальных
                          // координатах или единичный вектор
                          // направления, заданный пользователем

    VECTOR4D ux,uy,uz;     // Локальные оси для отслеживания
                          // полной ориентации. Значения этих
                          // переменных автоматически
                          // обновляются при вызове функций,
                          // осуществляющих поворот
    int num_vertices;      // Количество вершин в каркасе
```

```

// объекта
int num.. frames; // Количество кадров (новая
// переменная)
int total_vertices; // Общее количество вершин
int curr_frame; // Текущий кадр анимации. 0 - если
// кадр только один (новая
// переменная)

VERTEX4DTV1_PTR vlist_local; // Массив локальных вершин
VERTEX4DTV1_PTR vlist_trans; // Массив преобразованных
// вершин

// Эти указатели нужны, чтобы отслеживать положение
// начала текущего кадра в списке вершин, содержащемся в
// многокаркасных объектах
VERTEX4DTV1_PTR head_vlist_local;
VERTEX4DTV1_PTR head_vlist_trans;

// Список координат текстуры (новая переменная)
POINT2D_PTR tlist; // Максимальное количество равно
// 3*(число многоугольников)

BITMAP_IMAGE_PTR texture; // Указатель на данные о
// текстуре для ее простого
// отображения (новая
// переменная)

int num_polys; // Количество многоугольников в
// каркасе
POLY4DV2_PTR plist; // Указатель на многоугольники
// (новая переменная)

// МЕТОДЫ

// Правильно настроить каркас настолько важно, что для
// этого задается функция-член. Вызов функций без
// предварительной настройки может привести к
// неприятностям!
int Set_Frame(int frame);

> OBJECT4DV2, *OBJECT4DV2_PTR;

```

О предназначении большинства новых полей можно догадаться по их именам и приведенным выше определениям многоугольников. Почти все интересные изменения и дополнения связаны со списками вершин и координат текстуры, а также с поддержкой многокаркасных объектов. Приведем их описание.

- **\*avg\_radius, \*max\_radius.** Поскольку теперь в объектах может содержаться несколько каркасов, каждый каркас будет состоять из своего набора вершин. Таким образом, значения среднего и максимального радиусов должны храниться в массивах.
- **num\_frames.** Количество анимационных кадров.
- **curr\_frame.** Текущий активный каркас.

- **vlist\_local**. Указатель на набор вершин текущего каркаса.
- **vlist\_trans**. Указатель на набор вершин текущего каркаса (после преобразования вершин).
- **head\_vlist\_local**. Заголовок списка вершин.
- **head\_vlist\_trans**. Заголовок списка вершин (преобразованная версия).
- **tlist**. Указатель на заголовок списка координат текстуры. Для всех кадров, представляющих движение объекта, достаточно одного списка. Остановимся на минуту, чтобы убедиться в том, что мы хорошо понимаем, какая информация содержится в этом списке. Несмотря на то, что отображение текстуры еще не рассматривалось, основная особенность данного списка в том, что в нем содержатся фактические координаты текстуры для всех ячеек каркаса. Из геометрических соображений понятно, что всего имеется до  $3 * \text{num\_vertices}$  координат текстуры (т.е. утроенное количество вершин). С другой стороны, у таких каркасов, как куб, будут координаты текстуры  $(0,0)$ ,  $(0,1)$ ,  $(1,0)$  и  $(1,1)$ , т.е. всего четыре координаты. Таким образом, каждой вершине в данном массиве соответствует индекс, указывающий, какую координату текстуры следует использовать в данной вершине. В худшем случае для каждой вершины каждого многоугольника может понадобиться своя координата текстуры; в этом случае нужно будет использовать  $3 * \text{num\_vertices}$  координат текстуры.

Теперь поговорим об анимации и предназначении описанных выше полей. Как уже упоминалось, сначала будет предпринята попытка статической анимации, реализуемой с помощью многокаркасных объектов. Этот способ заключается в загрузке некоторого количества кадров и их последовательном отображении, как это происходит в двумерной анимации. Однако анимация каркасов будет производиться с некоторыми ограничениями. Во-первых, количество вершин для всех каркасов должно быть постоянным, а во-вторых, информация о вершинах, из которых состоит многоугольник, меняться не должна. Единственное, что допускает изменения, — это положения вершин в каркасе. Эти соглашения в основном нужны для того, чтобы обеспечить поддержку формата *Quake II* .MD2. В этой главе не будет реализована ни одна функция, предназначенная для анимации, однако будет проведена необходимая подготовительная работа.

#### СОВЕТ

Более развитые системы анимации поддерживают прямую или обратную кинематику, основанную не на информации о статических каркасах, а на физических моделях. Информация о вершинах каркаса преобразуется в соответствии с математической моделью; при этом учитываются внутренние связи объекта и действующие на него внешние силы или другие данные, полученные на предыдущих этапах анимации. Таким образом, чтобы создать полноценную анимацию, объект представляется в виде иерархической структуры автономно анимируемых подобъектов.

Поговорим еще немного о поддержке многокаркасных объектов. При выборе кадра используется указатель на его начало, количество содержащихся в соответствующем каркасе вершин и данные о текущем кадре. На основании этой информации определяется адрес нужного кадра. Затем в кадр передаются указатели **vlist\_local** и **vlist\_trans**, и получается, что все действия, которые выполняются в списках, соответствующих этим указателям, автоматически выполняются в нужном кадре. Описанный процесс проиллюстрирован на рис. 9.5. Единственное, о чем нужно позаботиться, — это убедиться в том, что с изменением каркаса корректно меняется его средний и максимальный радиус. Это значит, что должны быть доступны массивы, в которых хранится информация об этих ра-

диусах. Других изменений будет очень мало, поскольку все сводится к тривиальной манипуляции с указателями `vlist_local` и `vlist_trans`.

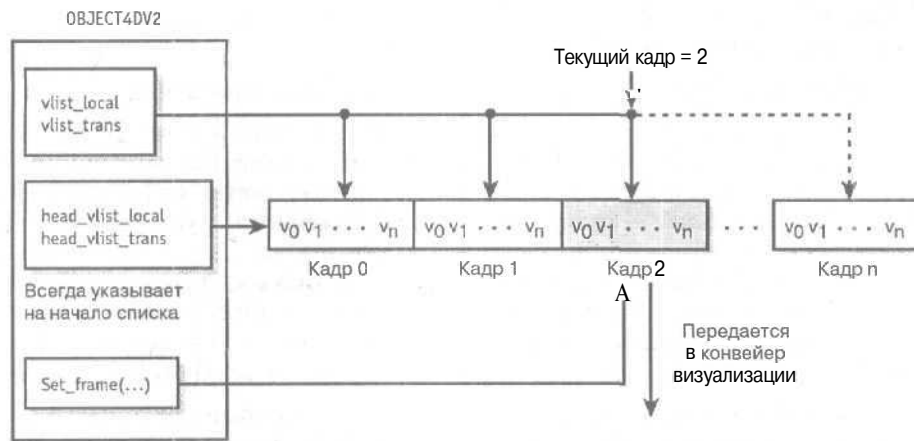


Рис. 9.5. Схема потока информации и преобразований многокадрных объектов

Заметим, что мы используем динамическое выделение памяти для массивов. Это позволяет экономнее расходовать память и облегчает поддержку многокадрных объектов. Единственная возникающая здесь проблема заключается в том, что не все форматы начинаются с чего-то наподобие указания количества многоугольников, вершин, координат текстур и т.п. А как было бы хорошо! Однако это не так, и поэтому при размещении массивов в памяти не обходится без небольших трюков (пару раз массивы даже приходится перемещать в памяти, считывая информацию из файлов). Например, сначала нужно создать объект `OBJECT4DV2`, а затем загрузить в него данные о вершинах и многоугольниках. Однако в формате `.COB` фактическое количество координат текстуры неизвестно заранее, поэтому приходится предусматривать худший вариант. В общем случае целесообразно придерживаться данного правила: если заранее неизвестно, сколько тех или иных объектов нужно разместить в памяти, выделите память для их максимально возможного количества; позже память можно будет перераспределить, скопировав нужные области и удалив неиспользованные.

В структуру `OBJECT4DV2` добавлен всего один метод, позволяющий задавать нужный кадр. Если в программе вызвать функцию, не установив предварительно кадр и не скопировав его надлежащим образом, это может привести к аварийному завершению ее работы. Имя метода — `Set_Frame()`, а его использование выглядит следующим образом.

```
object.Set_Frame(0);
```

В результате выполнения приведенной выше инструкции будет установлен кадр с индексом 0. Ознакомление с кодом метода `Set_Frame()` позволит увидеть, насколько просто в нем выполняются манипуляции с указателями. Приведем этот код.

```
// Задание кадра настолько важно, что для этого
// создается отдельный метод
int OBJECT4DV2::Set_Frame(int frame)
{
    // Эта функция устанавливает текущий кадр для
    // многокадрного объекта; если в объекте один кадр,
    // функция не выполняет никаких действий
```

```

// Проверка корректности объекта
if (!this)
    return(0);

// Проверка на многокаркасность
if (!(this->attr & OBJECT4DV2_ATTR_MULTI_FRAME))
    return (0);

// У нас имеется корректный объект

// Проверка границ кадра
if (frame < 0 )
    frame = 0;
else
    if (frame >= this->num_frames)
        frame = this->num_frames - 1;

// Установка кадра
this->curr_frame = frame;

// Обновление указателей с тем, чтобы они указывали на
// "блок" вершин, представляющих данный каркас;
// количество вершин во всех каркасах должно быть
// одинаковым, и они должны принадлежать одним и тем же
// многоугольникам, входящим в состав объекта. Вершины
// просто сдвигаются. Это нужно, чтобы перенаправить
// указатели на списки вершин на основе указателей на
// начала кадров

this->vlist_local = &(this->head_vlist_local[frame*this->
    num_vertices]);
this->vlist_trans = &(this->head_vlist_trans[frame*this->
    num_vertices]);

// Возврат кода успешного выполнения
return(1);
} // Set_Frame

```

#### НА ЗАМЕТКУ

Была также создана внешняя версия этой функции с именем `Set_OBJECT4DV2_Frame(OBJECT4DV2_PTR obj, int frame)`. Эта версия идентична методу `Set_Frame()`, однако ее можно вызывать как обычную функцию, а не как метод объекта.

Каждому объекту соответствует основная текстура, применяемая ко всему объекту в целом. Это означает, что хотя структуры, предназначенные для хранения данных о многоугольниках, поддерживают различные указатели на текстуру для каждого многоугольника, в объектах поддерживается только одна текстура. Возможно, со временем эта ситуация изменится, однако при текстурировании объектов удобно представлять текстуру как "обшивку". Таким образом, при загрузке объекта в него загружается единая текстурная карта. Затем вместе с многоугольниками загружаются внутренние указатели на соответствующие им фрагменты текстуры объекта `OBJECT4DV2`. Вот где пригодилась бы

библиотека материалов! С ее помощью можно было бы пользоваться несколькими текстурами, однако об этом будет сказано позже.

## Обзор списка функций и их прототипов

В этой главе было добавлено довольно много новых функций, поэтому стоит посвятить некоторое время их краткому обзору. Напомним, что большинство из них уже встречались, но теперь мы имеем дело со второй версией этих функций. Таким образом, будут перечислены все функции и приведено краткое объяснение их использования. Я не стану вдаваться в подробное описание параметров или приводить примеры, поскольку, как уже было сказано, это не новые функции, а всего лишь новые их версии. Поэтому они, в основном, нам уже знакомы. Просто сделаем итоговый обзор имеющихся программных инструментов.

Большинство приведенных ниже прототипов может вызвать небольшое недоумение, поскольку неизвестно, какие действия выполняют некоторые из новых функций. Однако все же есть смысл ознакомиться со всеми ними. Это даст общее представление об этапах игрового конвейера, что облегчит восприятие теоретического материала. Кроме того, добавлено несколько функций, не входящих в состав конечного игрового процессора. Они предназначены для наложения текстуры и поддержки структур OBJECT4DV1 и RENDERLIST4DV1. В данной главе эти функции не используются; они были написаны в целях тестирования при переходе к новой версии структур данных. Здесь эти функции не описываются, а ознакомиться с ними можно с помощью соответствующих файлов с расширениями .CPR и .H. Функции сгруппированы по их назначению.

### Прототип функции

```
char *Extract_Filename_From_Path(char *filepath,  
                                char *filename);
```

### Назначение

Извлекает имя файла из переданного полного пути к нему.

### Прототип функции

```
int Set_OBJECT4DV2_Frame(OBJECT4DV2_PTR obj, int frame);
```

### Назначение

Устанавливает указанный кадр анимации.

### Прототип функции

```
int Destroy_OBJECT4DV2(OBJECT4DV2_PTR obj);
```

### Назначение

Удаляет объект и освобождает выделенную для него память.

### Прототип функции

```
int Init_OBJECT4DV2(OBJECT4DV2_PTR obj, // Объект  
                   int _num_vertices,  
                   int _num_polys,  
                   int _num_frames,  
                   int destroy=0);
```

### Назначение

Выполняет инициализацию и выделение памяти для объекта.

### Прототип функции

```
void Translate_OBJECT4DV2(OBJECT4DV2_PTR obj,  
                          VECTOR4D_PTR vt);
```

**Назначение**

Выполняет перенос объекта в трехмерном пространстве.

**Прототип функции**

```
void Scale_OBJECT4DV2(OBJECT4DV2_PTR obj, VECTOR4D_PTR vs,
    int all_frames=0);
```

**Назначение**

Выполняет масштабное преобразование одного или всех каркасов объекта.

**Прототип функции**

```
void Transform_OBJECT4DV2(OBJECT4DV2_PTR obj,
    MATRIX4X4_PTR mt
    int coord_select,
    int transform_basis,
    int all_frames=0);
```

**Назначение**

Применяет заданную матрицу преобразования к текущему каркасу или ко всем каркасам объекта.

**Прототип функции**

```
void Rotate_XYZ_OBJECT4DV2(OBJECT4DV2_PTR obj,
    float theta_x,
    float theta_y,
    float theta_z,
    int all_frames);
```

**Назначение**

Поворачивает объекты вокруг осей x, y, z на заданные углы.

**Прототип функции**

```
void Model_To_World_OBJECT4DV2(OBJECT4DV2_PTR obj,
    int coord_select = TRANSFORM_LOCAL_TO_TRANS,
    int all_frames = 0);
```

**Назначение**

Выполняет преобразование переданного объекта из локальной системы координат в мировую.

**Прототип функции**

```
int Cull_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam,
    int cull_flags);
```

**Назначение**

Отбраковывает объект, а точнее говоря, удаляет его из поля зрения.

**Прототип функции**

```
void Remove_Backfaces_OBJECT4DV2(OBJECT4DV2_PTR obj,
    CAM4DV1_PTR cam);
```

**Назначение**

Удаляет задние поверхности указанного объекта, невидимые с точки зрения заданной камеры.

**Прототип функции**

```
void Remove_Backfaces_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, CAM4DV1_PTR cam);
```

**Назначение**

Удаляет из указанного списка визуализации задние поверхности, невидимые с точки зрения заданной камеры.

**Прототип функции**

```
void Camera_To_Perspective_OBJECT4DV2(OBJECT4DV2_PTR obj,
                                       CAM4DV1_PTR cam);
```

**Назначение**

Выполняет аксонометрическое преобразование указанного объекта, полученное с помощью заданной камеры.

**Прототип функции**

```
void Perspective_To_Screen_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_List CAM4DVI_PTR cam);
```

**Назначение**

Преобразует аксонометрические координаты в экранные.

**Прототип функции**

```
void Camera_To_Perspective_Screen_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, CAM4DV1_PTR cam);
```

**Назначение**

Выполняет преобразование указанного списка визуализации, полученного для заданной камеры, из системы отсчета камеры в систему отсчета экрана. По сути, эта функция представляет собой объединение функций `Camera_To_Perspective_RENDERLIST4DV2()` и `Perspective_To_Screen_RENDERLIST4DV2()`.

**Прототип функции**

```
void Camera_To_Perspective_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, CAM4DV1_PTR cam);
```

**Назначение**

Выполняет преобразование координат камеры в аксонометрические для указанного списка визуализации.

**Прототип функции**

```
void World_To_Camera_OBJECT4DV2(OBJECT4DV2_PTR obj,
                                  CAM4DV1_PTR cam);
```

**Назначение**

Выполняет переход из мировой системы отсчета в систему отсчета заданной камеры для указанного объекта.

**Прототип функции**

```
void World_To_Camera_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rendjist CAM4DVI_PTR cam);
```

**Назначение**

Выполняет переход из мировой системы отсчета в систему отсчета заданной камеры для указанного списка визуализации.

**Прототип функции**

```
void Camera_To_Perspective_Screen_OBJECT4DV2(
    OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

#### Назначение

Выполняет преобразование изображения указанного объекта, полученного с помощью заданной камеры, из системы координат камеры в систему координат экрана. По сути, эта функция представляет собой объединение функций `Camera_To_Perspective_OBJECT4DV2()` и `Perspective_To_Screen_OBJECT4DV2()`.

#### Прототип функции

```
void Perspective_To_Screen_OBJECT4DV2(OBJECT4DV2_PTR obj,  
                                       CAM4DV1_PTR cam);
```

#### Назначение

Преобразует аксонометрические координаты объекта в экранные.

#### Прототипы функций

```
int Insert_POLY4DV2_RENDERLIST4DV2(  
    RENDERLIST4DV2_PTR rend_list, POLY4DV2_PTR poly);
```

```
int Insert_POLYF4DV2_RENDERLIST4DV2(  
    RENDERLIST4DV2_PTR rendjlist, POLYF4DV2_PTR poly);
```

#### Назначение

Помешают в список визуализации указанный **многоугольник**.

#### Прототип функции

```
int Insert_OBJECT4DV2_RENDERLIST4DV2(  
    RENDERLIST4DV2_PTR rend_list,  
    OBJECT4DV2_PTR obj,  
    int insert_local);
```

#### Назначение

Помешает указанный объект в список визуализации, раскладывая его на многоугольники.

#### Прототип функции

```
void Reset_OBJECT4DV2(OBJECT4DV2_PTR obj);
```

#### Назначение

Устанавливает исходное состояние объекта.

#### Прототип функции

```
int Compute_OBJECT4DV2_Poly_Normals(OBJECT4DV2_PTR obj);
```

#### Назначение

Вычисляет нормали к многоугольнику для заданного каркаса, входящего в состав объекта; выполняет подготовку перед вызовом функций **освещения**.

#### Прототипы функций

```
void Draw_OBJECT4DV2_Wire(OBJECT4DV2_PTR obj,  
                          UCHAR *video_buffer, int lpitch);
```

```
void Draw_OBJECT4DV2_Wire16(OBJECT4DV2_PTR obj,  
                             UCHAR *video_buffer, int lpitch);
```

#### Назначение

Выводят на экран изображение указанного объекта в виде каркаса.

#### Прототипы функций

```
void Draw_RENDERLIST4DV2_Wire(RENDERLIST4DV2_PTR rendjlist,  
                               UCHAR *video_buffer, int lpitch);
```

```
void Draw_RENDERLIST4DV2_Wire16(
    RENDERLIST4DV2_PTR rendjist
    UCHAR *video_buffer, int lpitch);
```

#### Назначение

Выводят на экран изображение, хранящееся в указанном списке визуализации, в виде каркаса.

#### Прототипы функций

```
void Draw_RENDERLIST4DV2_Solid16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer, int lpitch);
```

```
void Draw_RENDERLIST4DV2_Solid(
    RENDERLIST4DV2_PTR rendjist
    UCHAR *video_buffer, int lpitch);
```

#### Назначение

Выводят на экран содержимое указанного списка визуализации; поддерживают многоугольники с излучательным и плоским затенением, а также текстурированные многоугольники.

#### Прототип функции

```
float Compute_OBJECT4DV2_Radius(OBJECT4DV2_PTR obj);
```

#### Назначение

Вычисляет средний и максимальный радиусы всех каркасов объекта.

#### Прототип функции

```
int Compute_OBJECT4DV2_Verx_Normals(OBJECT4DV2_PTR obj);
```

#### Назначение

Вычисляет нормали, проведенные в вершинах объекта.

#### Прототип функции

```
int Load_OBJECT4DV2_PLG(OBJECT4DV2_PTR obj, // Указатель
    // на объект
    char *filename, // Имя файла в формате PLG
    VECTOR4D_PTR scale, // Начальный масштаб
    VECTOR4D_PTR pos, // Начальное положение
    VECTOR4D_PTR rot, // Начальная ориентация
    int vertex_flags=0); // Флаги, управляющие поряд-
    // ком обхода вершин.
```

#### Назначение

Загружает с диска файл с трехмерным изображением в формате .PLG/PLX.

#### Прототип функции

```
int Load_OBJECT4DV2_3DSASC(OBJECT4DV2_PTR obj, // Указатель
    // на объект
    char *filename, // Имя файла в формате ASC
    VECTOR4D_PTR scale, // Начальный масштаб
    VECTOR4D_PTR pos, // Начальное положение
    VECTOR4D_PTR rot // Начальная ориентация
    int vertex_flags=0); // Флаги, управляющие поряд-
    // ком обхода вершин.
```

### Назначение

Загружает с диска файл с трехмерным изображением в формате 3D Studio Max .ASC; поддерживает изменение режимов затенения с помощью параметра `vertex_flags`.

### Прототип функции

```
int Load_OBJECT4DV2_COB(OBJECT4DV2_PTR obj, // Указатель
                        // на объект
                        char *filename, // Имя файла в формате
                        // Caligari COB
                        VECTOR4D_PTR scale, // Начальный масштаб
                        VECTOR4D_PTR pos, // Начальное положение
                        VECTOR4D_PTR rot, // Начальная ориентация
                        int vertex_flags=0); // Флаги, управляющие порядком
                                           // обхода вершин и параллельным
                                           // переносом.
```

### Назначение

Загружает с диска файл с трехмерным изображением в формате .COB; поддерживает объекты с постоянным и плоским затенением, а также затенением по Гуро и многоугольники с текстурой.

### Прототип функции

```
void Reset_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list);
```

### Назначение

Восстанавливает исходное состояние списка визуализации для последующего использования.

### Прототипы функций

```
int Light_OBJECT4DV2_World16(OBJECT4DV2_PTR obj, // Объект
                              CAM4DV1_PTR cam, // Расположение камеры
                              LIGHTV1_PTR Lights, // Список источников освещения
                                                    // (допускается один или
                                                    // несколько источников)
                              int max_lights); // Максимальное количество
                                                // источников в списке
```

```
int Light_OBJECT4DV2_World(OBJECT4DV2_PTR obj, // Объект
                            CAM4DV1_PTR cam, // Расположение камеры
                            LIGHTV1_PTR lights, // Список источников освещения
                                                  // (допускается один или
                                                  // несколько источников)
                            int max_lights); // Максимальное количество
                                              // источников в списке.
```

### Назначение

Освещают заданный объект с помощью указанной камеры и источников освещения; разработаны 8-битовая и 16-битовая версии этой функции.

### Прототипы функций

```
int Light_RENDERLIST4DV2_World16(
    RENDERLIST4DV2_PTR rend_list, // Список
    CAM4DV1_PTR cam, // Расположение камеры
    LIGHTV1_PTR Lights, // Список источников освеще-
```

```

int max_lights); // ния (допускается один или
                  // несколько источников)
                  // Максимальное количество
                  // источников в списке

int Light_RENDERLIST4DV2_World(
    RENDERLIST4DV2_PTR rend_list, // Список
    CAM4DV1_PTR cam,              // Расположение камеры
    LIGHTV1_PTR Lights,           // Список источников освеще-
    // ния (допускается один или
    // несколько источников)
    int max_lights);              // Максимальное количество
    // источников в списке.

```

#### Назначение

Освещают заданный список визуализации с помощью указанной камеры и источников освещения; разработаны 8-битовая и 16-битовая версии этой функции.

#### Прототип функции

```
void Sort_RENDERLIST4DV2(RENDERLIST4DV2_PTR rendlist,
    int sort_method);
```

#### Назначение

Выполняет с помощью указанного метода сортировку по оси z элементов, содержащихся в списке визуализации.

#### Прототипы функций

```
int Compare_AvgZ_POLYF4DV2(const void *arg1,
    const void *arg2);

int Compare_NearZ_POLYF4DV2(const void *arg1,
    const void *arg2);

int Compare_FarZ_POLYF4DV2(const void *arg1,
    const void *arg2);
```

#### Назначение

Методы сортировки; вызываются из функции Sort\_RENDERLIST4DV2().

#### Прототипы функций

```
void Draw_Textured_Triangle16(POLYF4DV2_PTR face,
    UCHAR *dest_buffer, int mem_pitch);

void Draw_Textured_Triangle(POLYF4DV2_PTR face,
    UCHAR *dest_buffer, int mem_pitch);
```

#### Назначение

Низкоуровневые функции визуализации, предназначенные для вывода треугольников с аффинной текстурой; существуют 8-битовая и 16-битовая версии этой функции.

#### Прототипы функций

```
void Draw_Textured_TriangleFS16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch);      // Количество байтов в строке
                        // (320, 640 и т.д.)
```

```
void Draw_Textured_TriangleFS(
    POLYF4DV2_PTR face,    // Указатель на поверхность
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch);        // Количество байтов в строке
                        // (320, 640 и т.д.).
```

#### Назначение

Низкоуровневые функции визуализации, предназначенные для вывода треугольников с плоским затенением; существуют 8-битовая и 16-битовая версии этой функции.

#### Прототипы функций

```
void Draw_Gouraud_Triangle16(
    POLYF4DV2_PTR face,    // Указатель на поверхность
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch);        // Количество байтов в строке
                        // (320, 640 и т.д.).
```

```
void Draw_Gouraud_Triangle(
    POLYF4DV2_PTR face,    // Указатель на поверхность
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch);        // Количество байтов в строке
                        // (320, 640 и т.д.).
```

#### Назначение

Низкоуровневые функции визуализации, предназначенные для вывода треугольников с затенением по Гуро; существуют 8-битовая и 16-битовая версии этой функции.

#### Прототипы функций

```
void Draw_Top_Tri2_16(float x1, float y1,
    float x2, float y2,
    float x3, float y3,
    int color,
    UCHAR *_dest_buffer, int mempitch);
```

```
void Draw_Bottom_Tri2_16(float x1, float y1,
    float x2, float y2,
    float x3, float y3,
    int color,
    UCHAR *_dest_buffer, int mempitch);
```

```
void Draw_Top_Tri2(float x1, float y1, float x2, float y2,
    float x3, float y3, int color,
    UCHAR *_dest_buffer, int mempitch);
```

```
void Draw_Bottom_Tri2(float x1, float y1,
    float x2, float y2,
    float x3, float y3,
    int color,
    UCHAR *_dest_buffer, int mempitch);
```

#### Назначение

Низкоуровневые функции визуализации, предназначенные для вывода треугольников с горизонтальной верхней и нижней сторонами; существуют 8-битовая и 16-битовая версии этих функций.

#### Прототипы функций

```
void Draw_Triangle_2D2_16(float x1, float y1,  
    float x2, float y2,  
    float x3, float y3,  
    int color,  
    UCHAR *dest_buffer, int mipmapitch);
```

```
void Draw_Triangle_2D2(float x1, float y1,  
    float x2, float y2,  
    float x3, float y3,  
    int color,  
    UCHAR *dest_buffer, int mipmapitch);
```

#### Назначение

Низкоуровневые функции визуализации, предназначенные для вывода треугольников общего вида; существуют 8-битовая и 16-битовая версии этой функции.

#### Прототип функции

```
int Load_Bitmap_File2(BITMAP_FILE_PTR bitmap,  
    char *filename);
```

#### Назначение

Функция для загрузки файлов с битовыми образами в формате .BMP и .eps в растровый файловый объект.

#### Прототип функции

```
int Load_Bitmap_PCX_File(BITMAP_FILE_PTR bitmap,  
    char *filename);
```

#### Назначение

Загружает файл в формате .eps в растровый файловый объект.

## Новые версии загрузчиков

Я рад, что мы вышли на уровень сложности, превышающий тот, который поддерживается некоторыми форматами файлов. В первую очередь это относится к форматам .PLG/PLX и .ASC, которые не поддерживают все то, что умеет делать наш игровой процессор. Поэтому в процессоре для этих форматов можно сделать не так уж много улучшений. Поддержка этих форматов игровым процессором сохраняется лишь потому, что с ними легко работать вручную. С другой стороны, формат Caligari .COB обладает достаточным количеством возможностей, поэтому здесь есть на что обратить внимание.

Как бы то ни было, формат .PLG/PLX поддерживает различные режимы затенения (ноне текстурные отображения). Кроме того, этот формат достаточно примитивен и подходит в основном для ручного моделирования черновых версий объектов для проверки их работы. Наконец, замечу, что жестко "прошитая" в формате .PLG поддержка 8- и 16-битового режимов причиняет много неудобств. Тем не менее, это хороший формат и он вполне подходит для начальной версии любого трехмерного игрового процессора.

Формат .ASC не позволяет работать почти ни с какими особенностями трехмерного объекта, кроме параметров каркаса и цвета. Этот формат не поддерживает ни отображения текстуры, ни моделей освещения — ничего. Поэтому я добавил в загрузчик возможность поддержки модели освещения. Однако и в формате .ASC, и в формате .PLG/PLX отсутствует поддержка текстуры, поскольку в них просто невозможно задать ее координаты.

ты. Кроме того, есть формат `.COB`, который поддерживает все эти возможности. Это единственный формат, в который из профаммы, создающей модели, можно экспортировать трехмерные объекты с информацией о текстуре и режиме освещения в том виде, который способен воспринять наш игровой процессор. Поэтому интуиция мне подсказывает, что по мере написания книги мы все больше внимания будем уделять формату `.COB` и все меньше — форматам `.PLG/PLX` и `.ASC`. С учетом вышесказанного, рассмотрим новые загрузчики, а также поддерживаемые ими особенности.

## Обновление загрузчика `.PLG/PLX`-файлов

В загрузчик файлов в формате `.PLG/PLX` было внесено наименьшее количество изменений. Однако это единственная функция, листинг которой я приведу в книге, поскольку он не такой большой, как другие. Основные изменения касаются поддержки новой структуры `OBJECT4DV2`, а также доступа к элементам данных и настройки некоторых новых флагов. Напомним, что новая структура `VERTEX4DTV1` содержит новые флаги, отвечающие за то, нужно ли задавать положение вершины, нормаль и координаты текстуры. Таким образом, я думаю, что полезно ознакомиться с кодом загрузчика файлов в формате `.PLG/PLX` с тем, чтобы понять некоторые из внесенных в него изменений. Это позволит лучше ориентироваться в коде других загрузчиков, который можно найти на прилагаемом компакт-диске. Ниже приводится код функции `Load_OBJECT4DV2_PLG()`; наиболее важные элементы в нем выделены полужирным шрифтом.

```
int Load_OBJECT4DV2_PLG(OBJECT4DV2_PTR obj,      // Указатель на
                        // объект
                        char*filename,             // Имя файла в формате .PLG
                        VECTOR4D_PTR scale,        // Начальный масштаб
                        VECTOR4D_PTR pas,          // Начальное положение
                        VECTOR4D_PTR rot           // Начальная ориентация
                        int vertex_flags)          // Флаги вершин, используемые
                                                // при перезаписи
{
    // Функция загружает с диска объекты в формате .PLG, а
    // также позволяет задавать в вызывающей программе масштаб,
    // положение и ориентацию объекта, чтобы избежать лишних
    // вызовов для статических объектов с одним каркасом
    // Поэтому при загрузке однокаркасного объекта OBJECT4DV2
    // следует надлежащим образом задавать поля функции

    FILE *fp;      // Указатель на файл
    char buffer[256]; // Рабочий буфер

    char*token_string; // Указатель на разбираемую лексему

    // Краткое описание формата; обратите внимание на тип в
    // конце каждого описания
    // # Комментарий
    // # Дескриптор объекта
    // имя_объекта количество_вершин количество_многоугольников
    // (строка) (целое число) (целое число)

    // # Список вершин
    // x0_float y0_float z0_float
```

```

// x1_float y1_float z1_float
// x2_float y2_float z2_float
//
//
// xn_float yn_float zn_float
//
// # Список многоугольников
// описание_поверхности_ushort кол_вершин_int
// индекс_v0_int индекс_v1_int ...
//
//
// описание_поверхности_ushort кол_вершин_int
// индекс_v0_int индекс_v1_int ...

// Для простоты считаем, что в каждой строке содержится
// по одному элементу. Таким образом, нужно найти дескриптор
// объекта, прочитать его, затем найти и считать список
// вершин и, наконец, список многоугольников

// Этап 1: обнуление и небольшая инициализация объекта
memset(obj, 0, sizeof(OBJECT4DV2));

// Задаем состояние объекта как активное и видимое
obj->state = OBJECT4DV2_STATE_ACTIVE |
            OBJECT4DV2_STATE_VISIBLE;

// Задаем положение объекта
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

// Задаем количество кадров
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;

// Этап 2: открываем файл
if (!(fp = fopen(filename, "r")))
{
    Write_Error("Couldn't open PLG file %s.", filename);
    return(0);
} // if

// Этап 3: извлекаем первую лексему, которой должен быть
// дескриптор объекта
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
{
    Write_Error("PLG file error with file %s"
                "(object descriptor invalid).", filename);
    return(0);
} // if

```

```

Write_Error("Object Descriptor: %s", token_string);

// Анализ информации об объекте
sscanf(token_string, "%s %d %d",
        obj->name, &obj->num_vertices, &obj->num_polys);

// Выделение памяти для вершин и многоугольников
// Некоторые параметры излишние, но это не имеет значения
if (!Init_OBJECT4DV2(obj, // Объект, помещаемый в память
        obj->num_vertices,
        obj->num_polys,
        obj->num_frames))
{
    Write_Error("\nPLG file error with file %s"
        " (can't allocate memory).", filename);
} // if

// Этап 4: загрузка списка вершин
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
{
    // Извлечение очередной вершины
    if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
    {
        Write_Error("PLG file error with file %s "
            "(vertex List invalid).", filename);
        return(0);
    } // if

    // Анализ вершины
    sscanf(token_string, "%f %f %f %f",
        &obj->vlist_local[vertex].x,
        &obj->vlist_local[vertex].y,
        &obj->vlist_local[vertex].z);
    obj->vlist_local[vertex].w = 1;

    // Масштабное преобразование вершин
    obj->vlist_local[vertex].x*=scale->x;
    obj->vlist_local[vertex].y*=scale->y;
    obj->vlist_local[vertex].z*=scale->z;

    Write_Error("\nVertex %d = %f, %f, %f, %f", vertex,
        obj->vlist_local[vertex].x,
        obj->vlist_local[vertex].y,
        obj->vlist_local[vertex].z,
        obj->vlist_local[vertex].w);

    // Для каждой вершины, как минимум, задается положение;
    // вид необходимой информации устанавливается с помощью
    // флагов
    SET_BIT(obj->vlist_local[vertex].attr,
        VERTEX4DTV1_ATTR_POINT);
}

```

```

} // for vertex

// Вычисление среднего и максимального радиусов
Compute_OBJECT4DV2_Radius(obj);

Write_Error("\nObject average radius - %f, "
           "max radius = %f",
           obj->avg_radius, obj->max_radius);

int poly_surface_desc = 0; // Дескриптор поверхности
                          // в формате PLG/PLX
int poly_num_verts = 0;   // Количество вершин в текущем
                          // многоугольнике (их всегда 3)
char tmp_string[8];       // Временная переменная для хранения
                          // дескриптора поверхности; проверяем,
                          // нужно ли преобразовать его из
                          // шестнадцатеричной системы счисления

// Этап 5: загрузка списка многоугольников
for (int poly=0; poly < obj->num_polys; poly++)
{
    // Извлечение дескриптора очередного многоугольника
    if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
    {
        Write_Error("PLG file error with file %s "
                    "(polygon descriptor invalid).", filename);
        return(0);
    } // if

    Write_Error("\nPolygon %d:", poly);

    // В каждом списке вершин ДОЛЖНО содержаться 3 вершины,
    // поскольку мы придерживаемся правила, что все модели
    // должны состоять из треугольников; считываем
    // дескриптор поверхности, количество вершин и их список
    sscanf(token_string, "%s %d %d %d", tmp_string,
           &poly_num_verts, // Всегда должно равняться 3
           &obj->plist[poly].vert[0],
           &obj->plist[poly].vert[1],
           &obj->plist[poly].vert[2]);

    // Поскольку дескриптор поверхности может быть в
    // шестнадцатеричном формате (когда в его начале
    // стоят символы "0x"), нужно провести проверку
    if (tmp_string[0] == '0' &&
        toupper(tmp_string[1]) == 'X')
        sscanf(tmp_string, "%x", &poly_surface_desc);
    else
        poly_surface_desc = atoi(tmp_string);

    // Указываем из списка вершин многоугольника на список
    // вершин объекта. Заметим, что это излишне, поскольку
    // список многоугольников в данном случае содержится в

```

```
// объекте, и пользователь волен выбрать, какой список
// используется при построении многоугольника -
// локальный или преобразованный. Для многоугольников,
// входящих в состав объекта, для этого параметра лучше
// задать значение NULL
obj->plist[poly].vlist = obj->vlist_local;
```

```
Write_Error( "\nSurface Desc - 0x%.4x, num_verts - %d, "
    "vert_indices [%d, %d, %d]",
    poly_surface_desc,
    poly_num_verts,
    obj->plist[poly].vert[0],
    obj->plist[poly].vert[1],
    obj->plist[poly].vert[2]);
```

```
// Теперь, когда в многоугольник загружены список
// вершин, а также их индексы, проанализируем дескриптор
// поверхности и на его основе зададим поля
// многоугольника
```

```
// Извлекаем из дескриптора поверхности все поля;
// сначала решим вопросы, связанные с односторонностью
// или двусторонностью поверхностей
if ((poly_surface_desc & PLX_2SIDED_FLAG))
{
    SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_2SIDED);
    Write_Error("\n2 sided.");
} // if
else
{
    // Односторонняя поверхность
    Write_Error("\n1 sided.");
} // else
```

```
// Зададим тип и значение цвета
if ((poly_surface_desc & PLX_COLOR_MODE_RGB_FLAG))
{
    // Поверхность в режиме RGB 4.4.4
    SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_RGB16);

    // Извлекаем цвет и копируем его в переменную,
    // хранящую цвет многоугольника в надлежащем
    // 16-битовом формате. Это формат 0x0RGB, в котором
    // на каждый пиксель отводится по 4 бита
    int red = ((poly_surface_desc & 0x0f00) >> 8);
    int green = ((poly_surface_desc & 0x00f0) >> 4);
    int blue = (poly_surface_desc & 0x000f);

    // Данные представляются в формате 4.4.4, а
    // графическая карта работает либо в формате 5.5.5,
    // либо в формате 5.6.5. Наша виртуальная система
    // формирования цвета преобразует формат 8.8.8 в
    // формат 5.5.5 или в формат 5.6.5. Однако перед
```

```

// ЭТИМ нужно преобразовать все значения, заданные в
// формате 4.4.4, в формат 8.8.8
obj->plist[poly].color = RGB16Bit(red*16, green*16,
                                blue*16);
Write_Error("\nRGB color = [%d, %d, %d]", red,
            green, blue);
} // if
else
{
    // Цвет поверхности задан в 8-битовом режиме
    SET_BIT(obj->plist[poly].attr,
            POLY4DV2_ATTR_8BITCOLOR);

    // Просто извлекаем последние 8 битов;
    // это и будет индекс цвета
    obj->plist[poly].color = (poly_surface_desc & 0x00ff);

    Write_Error("\n8-bit color index = %d",
                obj->plist[poly].color);
} // else

// Обработка режима затенения
int shade_mode = (poly_surface_desc & PLX_SHADE_MODE_MASK);

// Задаем режим затенения многоугольника
switch (shade_mode)
{
    case PLX_SHADE_MODE_PURE_FLAG: {
        SET_BIT(obj->plist[poly].attr,
                POLY4DV2_ATTR_SHADE_MODE_PURE);
        Write_Error("\nShade mode - pure");
    } break;

    case PLX_SHADE_MODE_FLAT_FLAG: {
        SET_BIT(obj->plist[poly].attr,
                POLY4DV2_ATTR_SHADE_MODE_FLAT);
        Write_Error("\nShade mode = flat");
    } break;

    case PLX_SHADE_MODE_GOURAUD_FLAG: {
        SET_BIT(obj->plist[poly].attr,
                POLY4DV2_ATTR_SHADE_MODE_GOURAUD);

        // Для этого многоугольника необходимо
        // определить нормали, проведенные во всех его
        // вершинах Устанавливаем это с помощью флагов
        SET_BIT(obj->vlist_local[obj->plist[poly]
                .vert[0]].attr, VERTEX4DTV1_ATTR_NORMAL);
        SET_BIT(obj->vlist_local[obj->plist[poly]
                .vert[1]].attr, VERTEX4DTV1_ATTR_NORMAL);
        SET_BIT(obj->vlist_local[obj->plist[poly]
                .vert[2]].attr, VERTEX4DTV1_ATTR_NORMAL);
    }
}

```

```

        .vert[2]].attr, VERTEX4DTV1_ATTR_NORMAL);

    Write_Error("\nShade mode = gouraud");
} break;

case PLX_SHADE_MODE_PHONG_FLAG: {
    SET_BIT(obj->plist[poly].attr,
        POLY4DV2_ATTR_SHADE_MODE_PHONG);

    // Для этого многоугольника необходимо
    // определить нормали, проведенные во всех его
    // вершинах Устанавливаем это с помощью флагов
    SET_BIT(obj->vlist_local[obj->plist[poly]
        .vert[0]].attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[obj->plist[poly]
        .vert[1]].attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[obj->plist[poly]
        .vert[2]].attr, VERTEX4DTV1_ATTR_NORMAL);

    Write_Error("\nShade mode = phong");
} break;

default: break;
} // switch

// Задаем версию 1.0 эмуляции материала
SET_BIT(obj->plist[poly].attr,
    POLY4DV2_ATTR_DISABLE_MATERIAL);

// Задаем активное состояние многоугольника
obj->plist[poly].state = POLY4DV2_STATE_ACTIVE;

// Указываем из списка вершин многоугольника на список
// вершин объекта. Заметим, что это излишне, поскольку
// список многоугольников в данном случае содержится в
// объекте, и пользователь волен выбрать, какой список
// используется при построении многоугольника -
// локальный или преобразованный. Для многоугольников,
// входящих в состав объекта, для этого параметра лучше
// задать значение NULL
obj->plist[poly].vlist = obj->vlist_local;

// Задаем список координат текстуры; необходимый этап
obj->plist[poly].tlist = obj->tlist;

} // for poly

// Вычисляем длины нормалей к многоугольникам
Compute_OBJECT4DV2_Poly_Normals (obj);

// Вычисляем нормали в вершинах для всех многоугольников
// с затенением по Гуро

```

**Compute \_OBJECT4DV2\_Vertex\_Normals (obj);**

```
// Закрываем файл  
fclose(fp);
```

```
// Возвращаем код успешного выполнения  
return (1);
```

```
} // Load_OBJECT4DV2_PLG
```

Рассмотрим подробнее все фрагменты кода, выделенные полужирным шрифтом. С помощью всех этих фрагментов реализованы новые особенности загрузчика. Первым в функции встречается следующий выделенный фрагмент.

```
// Задаем количества кадров  
obj->num_frames = 1;  
obj->curr_frame = 0;  
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;
```

Он нужен для поддержки объектов с несколькими каркасами и установки нулевого кадра. Это также нужно выполнять для объектов с одним каркасом. Далее следует вызов функции, выделяющей память для объекта **OBJECT4DV2**.

```
if (!Init_OBJECT4DV2(obj, // Объект, помещаемый в память  
    obj->num_vertices,  
    obj->num_polys,  
    obj->num_frames))
```

```
{  
    Write_Error("\nPLG file error with file %s "  
        "(can't allocate memory).", filename);  
}
```

```
// if
```

Эта функция только выделяет область памяти для объекта или удаляет ее, если она уже выделена. Ниже приведен листинг функции **Init\_OBJECT4DV2()**.

```
int Init_OBJECT4DV2(OBJECT4DV2_PTR obj, // Объект, для  
    // которого выделяется память
```

```
    int _num_vertices,  
    int _num_polys,  
    int _num_frames,  
    int destroy)
```

```
{  
    // Функция выделяет память для объекта OBJECT4DV2 на  
    // основе переданных в нее параметров. Возможно,  
    // впоследствии будет создана более сложная версия,  
    // однако пока что мы не хотим нагружать функцию  
    // инициализации другими задачами. В 99% случаев все  
    // будет производиться путем вызова функции,  
    // предназначенной для загрузки объекта. Возможно, эта  
    // функция понадобится при создании объекта вручную
```

```
    // Если объект существует, сначала он удаляется
```

```
    if (destroy)
```

```
        Destroy_OBJECT4DV2(obj);
```

```

// Выделение памяти для списка вершин
if (!(obj->vlist_local = (VERTEX4DTV1_PTR)
    malloc(sizeof(VERTEX4DTV1)*
        _num_vertices*_num_frames)))
    return (0);

// Очистка данных
memset((void *)obj->vlist_local,0,sizeof(VERTEX4DTV1)*
    _num_vertices*_num_frames);

if (!(obj->vlist_trans = (VERTEX4DTV1_PTR)
    malloc(sizeof(VERTEX4DTV1)*
        _num_vertices*_num_frames)))
    return (0);

// Очистка данных
memset((void *)obj->vlist_trans,0,sizeof(VERTEX4DTV1)*
    _num_vertices*_num_frames);

// Количество координат текстуры равно
// 3*(число многоугольников)
if (!(obj->tlist = (POINT2D_PTR)
    malloc(sizeof(POINT2D)*_num_polys*3)))
    return(0);

// Очистка данных
memset((void *)obj->tlist,0,
    sizeof(POINT2D)*_num_polys*3);

// Выделение памяти для массива со значениями радиусов
if (!(obj->avg_radius =
    (float *)malloc(sizeof(float)*_num_frames)))
    return(0);

// Очистка данных
memset((void *)obj->avg_radius,0,
    sizeof(float)*_num_frames);

if (!(obj->max_radius =
    (float *)malloc(sizeof(float)*_num_frames)))
    return(0);

// Очистка данных
memset((void *)obj->max_radius,0,
    sizeof(float)*_num_frames);

// Выделение памяти для списка вершин
if (!(obj->plist = (POLY4DV2_PTR)
    malloc(sizeof(POLY4DV2)*_num_polys)))
    return(0);

// Очистка данных

```

```

memset((void *)obj->plist,0,
        sizeof(POLY4DV2)*_num_polys);

// Псевдонимы указателей на начало списка
obj->head_vlist_local = obj->vlist_local;
obj->head_vlist_trans = obj->vlist_trans;

// Присвоение значений внутренним переменным
obj->num_frames = _num_frames;
obj->num_polys = _num_polys;
obj->num_vertices = _num_vertices;
obj->total_vertices = _num_vertices*_num_frames;

// Возврат кода успешного выполнения
return(1);
} // Init_OBJECT4DV2

Возможно, вы обратили внимание, что в этой функции вызывается другая функция,
Destroy_OBJECT4DV2(), код которой приведен ниже.
int Destroy_OBJECT4DV2(OBJECT4DV2_PTR obj) // Удаляемый
                                           // объект
{
    // Функция удаляет переданный в нее объект, освобождая
    // выделенную для него память (если память была
    // выделена)

    // Список локальных вершин
    if (obj->head_vlist_local)
        free(obj->head_vlist_local);

    // Список преобразованных вершин
    if (obj->head_vlist_trans)
        free(obj->head_vlist_trans);

    // Список координат текстуры
    if (obj->tlist)
        free(obj->tlist);

    // Список многоугольников
    if (obj->plist)
        free(obj->plist);

    // Массивы, содержащие радиусы объекта
    if (obj->avg_radius)
        free(obj->avg_radius);

    if (obj->max_radius)
        free(obj->max_radius);

    // Окончательное удаление объекта
    memset((void *)obj, 0, sizeof(OBJECT4DV2));
}

```

```
// Возврат кода успешного выполнения
return(1);
```

```
} // Destroy_OBJECT4DV2
```

Продолжим рассмотрение загрузчика файлов в формате .PLG. С помощью приведенного ниже важного фрагмента кода устанавливается надлежащее значение поля всех вершин, для которых должно задаваться положение.

```
// Для каждой вершины, как минимум, задается положение;
// вид необходимой информации устанавливается с помощью
// флагов
SET_BIT(obj->vlist_local[vertex].attr,
        VERTEX4DTV1_ATTR_POINT);
```

Это действие нужно выполнить для всех вершин, для которых определено положение (т.е. вообще почти для всех вершин). Несмотря на кажущуюся нелепость этого фрагмента, он является частью новой системы, в которой определяется *гибкий формат вершин*, позволяющий задавать для них положение, нормали и координаты текстуры. Несколько строками ниже можно увидеть фрагменты кода, с помощью которых реализуется затенение по Гуро и Фонгу. Эти фрагменты позволяют помещать в блок данных, описывающий вершины, информацию о нормалях.

```
// Для этого многоугольника необходимо определить
// нормали, проведенные во всех его вершинах.
// Делаем это с помощью соответствующих флагов
SET_BIT(obj->vlist_local[obj->plist[poly].vert[0]].attr,
        VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[obj->plist[poly].vert[1]].attr,
        VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[obj->plist[poly].vert[2]].attr,
        VERTEX4DTV1_ATTR_NORMAL);
```

Нормали к элементам поверхностей, проведенные в каждой вершине многоугольника, понадобятся для его затенения по Гуро и по Фонгу (не исключено, что последний вид затенения реализовываться не будет), поэтому здесь устанавливаются надлежащие атрибуты. Наконец, вызываются две функции, вычисляющие нормали к многоугольникам и нормали, проведенные в вершине и усредненные по всем многоугольникам, для которых данная вершина является общей. Функция, вычисляющая нормаль к многоугольнику, нам уже встречалась.

```
// Вычисляем длины нормалей к многоугольникам
Compute_OBJECT4DV2_Poly_Normals(obj);
```

А это новая функция, которая вычисляет параметры усредненных нормалей к каждому многоугольнику, проведенных в его вершинах.

```
// Вычисляем нормали в вершинах для всех многоугольников
// с затенением по Гуро
Compute_OBJECT4DV2_Vertex_Normals(obj);
```

Обсудим эту функцию подробнее.

## Вычисление нормалей для реализации затенения

В следующем разделе будут изложены принципы освещения, а также алгоритм затенения по Гуро. Однако давайте забудем об этом на время и сосредоточим внимание на

том, как вычислить нормали, проведенные в вершинах многоугольников. Перед нами стоит такая задача: имеется каркас модели, состоящий из  $p$  многоугольников и  $v$  вершин, причем для каждой из этих вершин  $v_i$  нужно вычислить соответствующую ей нормаль. На первый взгляд все понятно, но возникает вопрос: о каких именно нормалях идет речь? Дело в том, что для моделирования освещения понадобятся усредненные нормали. На рис. 9.6 изображен простой каркас и нормали, проведенные в каждой вершине каждого многоугольника. Для вычисления параметров таких нормалей нужно определить, в состав каких многоугольников она входит, а затем провести усреднение всех нормалей, проведенных к этим многоугольникам. Более того, при усреднении придется принимать во внимание площадь каждого многоугольника, смежного с данной вершиной.

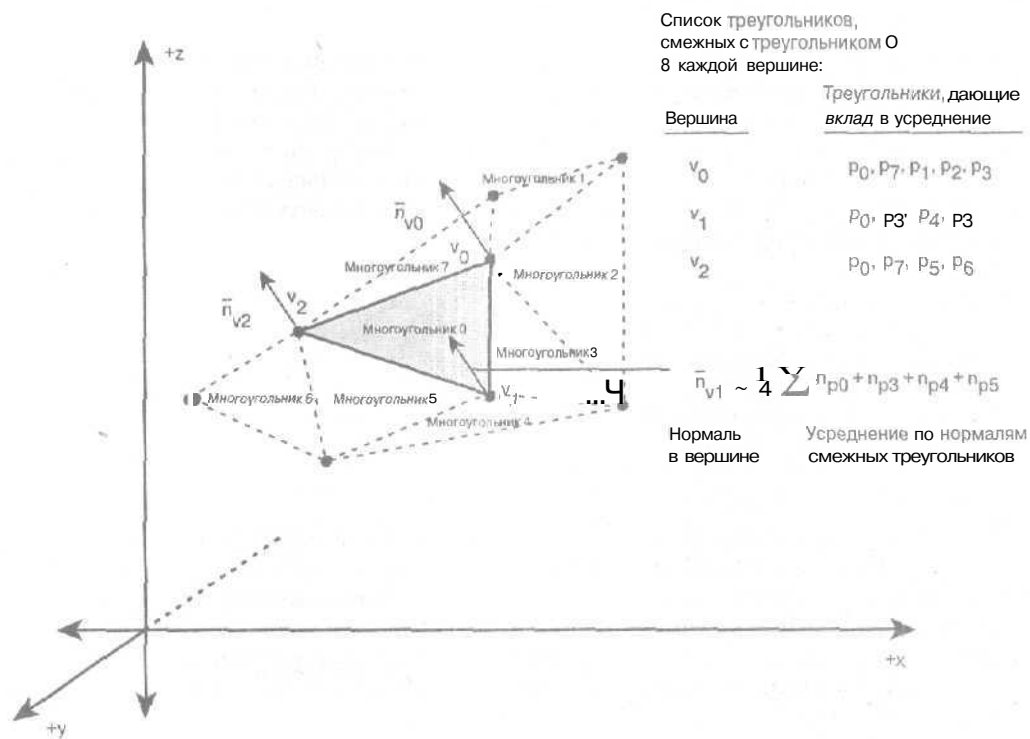


Рис. 9.6. Вычисление нормалей в вершинах с учетом вкладов многоугольников

Например, если есть два многоугольника с общей вершиной, как показано на рис. 9.7, то нормаль к многоугольнику 1 должна давать более весомый вклад в усредненную нормаль в вершине, чем нормаль к многоугольнику 2. Это объясняется тем, что площадь многоугольника 1 намного превышает площадь многоугольника 2. По сути, кодируется это тривиально. Используемый при этом трюк состоит в том, что если нормаль к треугольнику вычисляется как векторное произведение векторов, отложенных вдоль сторон этого треугольника, то ее длина равна удвоенной площади треугольника. Другими словами, длина нормали равна площади параллелограмма, построенного на сторонах треугольника так, как показано на рис. 9.8. Это видно из такого соотношения:

$$|u \times v| = (\text{основа}) \times (\text{высота}) = \text{площадь параллелограмма}.$$

Таким образом,  $|u \times v|/2$  — это площадь треугольника, образованного векторами  $u$  и  $v$ .  
 Нормаль, проведенная в вершине, будет вычисляться по следующему алгоритму.

```

для каждой вершины каркаса  $v_i$ 
begin
     $v\_sum\_normal = \langle 0, 0, 0 \rangle$ 
    для каждого многоугольника  $p_i$ ,
        смежного с вершиной  $v_i$ 
    begin
         $v\_sum\_normal += p_i\_normal$ 
    end
end
нормируем  $v\_sum$ 
    
```

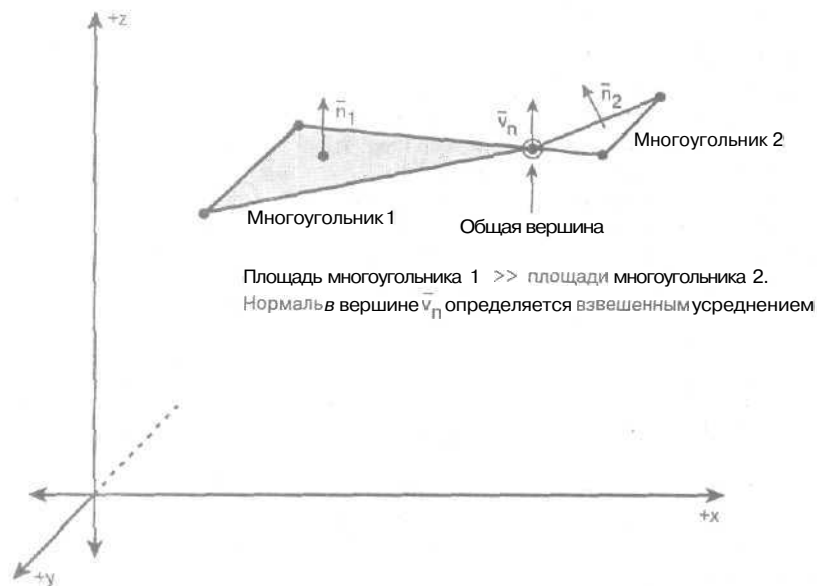


Рис. 9.7. При усреднении нормали в вершине вклад каждой нормали к многоугольнику пропорционален его площади

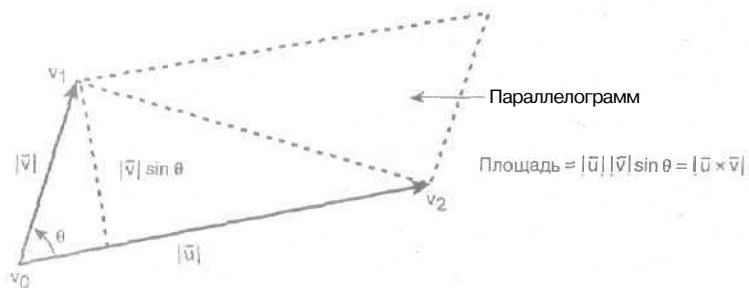


Рис. 9.8. Модуль векторного произведения равен площади параллелограмма, построенного на этих векторах

Далее начинается сложная часть. После того как все вычисления выполнены, возникает **проблема**: есть вектор нормали, проведенный в вершине, однако в процессе преобразования каркаса необходимо **выполнить** и преобразование этого вектора, чтобы поддерживать целостность модели! Это важно; в противном случае нормали окажутся бесполезными. К **сожалению**, при этом для каждой вершины приходится выполнять дополнительные **операции**. Другими словами, возникает необходимость преобразования вектора нормали, в результате чего количество вершин на многоугольник с затенением по Гуро **удваивается**. Однако это относится только к многоугольникам, которые освещаются по методу Гуро, поэтому дополнительные затраты ресурсов сводятся к минимуму, если таких многоугольников мало или если их нет.

Принимая во внимание сказанное, разработаем функцию, вычисляющую нормали в вершинах во время загрузки объекта. Ее код приведен ниже.

```
Compute_OBJECT4DV2_Verex_Normals(OBJECT4DV2_PTR obj)
{
    // Информация о нормалях в вершинах каждого
    // многоугольника используется в нескольких функциях,
    // однако важнее всего она для функции, моделирующей
    // освещение многоугольников с затенением по Гуро. Таким
    // образом, нормали нужны только для вершин
    // многоугольников с затенением по Гуро. Для каждой
    // вершины следует определить, в состав каких
    // многоугольников она входит, а затем усреднить нормали
    // всех этих многоугольников. Определить, вносит ли
    // вклад тот или иной многоугольник, можно на основе его
    // флагов затенения

    // Проверяем, корректен ли объект
    if (!obj)
        return(0);

    // Алгоритм: просмотрим список многоугольников и соберем
    // информацию о нормалях всех многоугольников, смежных с
    // данным. С помощью счетчиков определяется количество
    // многоугольников, которые вносят вклад в данную
    // вершину. После того, как мы дойдем до конца списка,
    // значения этих счетчиков будут использоваться в
    // операции усреднения по отмеченным вершинам. Таким
    // образом, вместо алгоритма, сложность которого
    // возрастает по закону  $O(n^2)$ , получаем алгоритм,
    // сложность которого возрастает по закону  $O(c*n)$ 

    // Отслеживаем индексы многоугольников, к которым
    // относится данная вершина. Этот объект используется
    // для подсчета количества многоугольников, дающих вклад
    // в эту вершину. В конце процесса "суммарная" нормаль
    // делится на количество многоугольников и усредняется
    int polys_touch_vertex[OBJECT4DV2_MAX_VERTICES];
    memset((void *)polys_touch_vertex, 0,
           sizeof(int)*OBJECT4DV2_MAX_VERTICES);

    // Просматриваем список многоугольников, из которых
    // состоит объект, вычисляем нормали к ним, а затем
```

```

// заносим каждую вершину, входящую в состав
// многоугольника, в массив "соприкасающихся" вершин.
// Это происходит в процессе аккумуляции информации
// о нормали в массиве нормалей, проведенных в вершинах

```

```

for (int poly=0; poly<obj->num_polys; poly++)
{
    Write_Error("\nprocessing poly %d", poly);

    // Проверяем, нужна ли в вершинах данного
    // многоугольника информация о нормалях
    if (obj->plist[poly].attr &
        POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
    {
        // Извлекаем индексы вершин в основной список
        // Напомним, что многоугольники не являются
        // самодостаточными; они основаны на списке
        // вершин, содержащемся в самом объекте
        int vindex_0 = obj->plist[poly].vert[0];
        int vindex_1 = obj->plist[poly].vert[1];
        int vindex_2 = obj->plist[poly].vert[2];

        Write_Error("\nTouches vertices: %d, %d, %d",
            vindex_0, vindex_1, vindex_2);

        // Нужно вычислить нормаль к поверхности этого
        // многоугольника с учетом того, что вершины
        // обходятся по часовой стрелке, u=p0->p1,
        // v=p0->p2, n=uxv
        VECTOR4D u, v, n;

        // Определяем векторы u и v
        VECTOR4D_Build(&obj->vlist_local[vindex_0].v,
            &obj->vlist_local[vindex_1].v,
            &u);
        VECTOR4D_Build(&obj->vlist_local[vindex_0].v,
            &obj->vlist_local[vindex_2].v,
            &v);

        // Вычисляем векторное произведение
        VECTOR4D_Cross(&u, &v, &n);

        // Обновляем массив вершин, при котором данный
        // многоугольник отмечается как вносящий свой
        // вклад
        polys_touch_vertex[vindex_0]++;
        polys_touch_vertex[vindex_1]++;
        polys_touch_vertex[vindex_2]++;

        Write_Error("\nPoly touch array v[%d] - %d, "
            "v[%d] = %d, v[%d] = %d",
            vindex_0,
            polys_touch_vertex[vindex_0],

```

```

        vindex_1, polys_touch_vertex[vindex_1],
        vindex_2, polys_touch_vertex[vindex_2]);

    // Суммируем все нормали, которые относятся к
    // данной вершине, в соответствующей переменной.
    // Заметим, что нормали не нормируются, так как
    // их длина играет роль при усреднении (длина
    // нормали равна удвоенной площади треугольника)
    VECTOR4D_Add(&obj->vlist_local[vindex_0].n, &n,
        &obj->vlist_local[vindex_0].n);
    VECTOR4D_Add(&obj->vlist_local[vindex_1].n, &n,
        &obj->vlist_local[vindex_1].n);
    VECTOR4D_Add(&obj->vlist_local[vindex_2].n, &n,
        &obj->vlist_local[vindex_2].n);
} // for poly

} // if

// Почти все готово. Собрана информация о нормалях во
// всех вершинах. Осталось только все усреднить
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
{
    // Если для данной вершины внесен хоть какой-нибудь
    // вклад, нужно провести усреднение. Можно также
    // проверить значения флагов, отвечающих за вид
    // затенения. Оба метода должны давать один и тот
    // же результат
    Write_Error("\nProcessing vertex:%d, attr:%d, "
        "contributors:%d", vertex,
        obj->vlist_local[vertex].attr,
        polys_touch_vertex[vertex]);

    // Проверяем, нужно ли в этой вершине вычислять
    // нормаль и выполнять усреднение
    if (polys_touch_vertex[vertex] >= 1)
    {
        obj->vlist_local[vertex].nx/=
            polys_touch_vertex[vertex];
        obj->vlist_local[vertex].ny/=
            polys_touch_vertex[vertex];
        obj->vlist_local[vertex].nz/=
            polys_touch_vertex[vertex];

        // Нормируем вектор нормали
        VECTOR4D_Normalize(&obj->vlist_local[vertex].n);

        Write_Error("\nAvg Vertex normal: [%f, %f, %f]",
            obj->vlist_local[vertex].nx,
            obj->vlist_local[vertex].ny,
            obj->vlist_local[vertex].nz);
    } // if
}

```

```

} // for

// Возврат кода успешного выполнения
return (1);

} // Compute_OBJECT4DV2_Vertex_Normals

```

При запуске любой из демонстрационных программ в файл `ERROR.TXT` выводится определенная информация. Возможно, вам захочется заглянуть в этот файл и кратко ознакомиться с фазой анализа вершины — это дает представление о том, как работает алгоритм.

## Обновление загрузчика файлов в формате .ASC

Загрузчик файлов в формате `.ASC` не так уж сильно изменился по сравнению с предыдущей версией, поскольку он не поддерживает текстуры, освещение и т.д. Единственная особенность, добавленная в эту функцию, — поддержка функциональных возможностей по обработке объекта `OBJECT4DV2`, описанных для загрузчика файлов в формате `.PLG/PLX`. Однако чтобы вдохнуть в этот формат немного жизни, я решил реализовать возможность, которую я называю *перезаписью вершин* (*vertex overrides*). Она позволяет внедрить или выбрать модель освещения, по крайней мере, на уровне объекта. Возможно, впоследствии мы добавим *глобальное текстурирование*, реализуемое с помощью некоторых обобщенных координат текстуры, которые будут работать для простых геометрических примитивов. Функция слишком объемная, поэтому полностью ее листинг не приводится. Как уже было сказано, добавления в ней аналогичны добавлениям, сделанным в загрузчике файлов в формате `.PLG/PLX`, которые связаны с инициализацией объекта и заданием типа вершин. Ниже приведен прототип функции-загрузчика файлов в формате `.PLG/PLX`.

```

int Load_OBJECT4DV2_3DSASC(OBJECT4DV2_PTR obj,
                           // Указатель на объект
                           char *filename, // Имя файла в формате ASC
                           VECTOR4D_PTR scale, // Начальный масштаб
                           VECTOR4D_PTR pos, // Начальное положение
                           VECTOR4D_PTR rot // Начальная ориентация
                           int vertex_flags) // Флаги, предназначенные для
                                           // изменения порядка вершин и
                                           // их перезаписи для затенения

```

Все новые функциональные возможности основаны на перезаписи информации об освещении вершин, как показано ниже.

```

// Новая часть. Используется в простых форматах для пере-
// записи и управления информацией, касающейся освещения
#define VERTEX_FLAGS_OVERRIDE_MASK 0xf000 // Маска для
                                           // извлечения перезаписанной информации

#define VERTEX_FLAGS_OVERRIDE_CONSTANT 0x1000
#define VERTEX_FLAGS_OVERRIDE_EMISSIVE 0x1000 // Псевдоним
#define VERTEX_FLAGS_OVERRIDE_PURE 0x1000
#define VERTEX_FLAGS_OVERRIDE_FLAT 0x2000
#define VERTEX_FLAGS_OVERRIDE_GOURAUD 0x4000
#define VERTEX_FLAGS_OVERRIDE_TEXTURE 0x8000

```

Как видим, есть возможность перезаписывать почти любые данные. Из приведенного фрагмента кода видно, что на данном этапе есть возможность провести перезапись только для чистого (*излучательного*) затенения, плоского затенения, а также затенения по Гуро.

```
// Сначала проверяем, производилась ли перезапись вообще
```

```
int vertex_overrides = (vertex_flags &  
    VERTEX_FLAGS_OVERRIDE_MASK);
```

```
if (vertex_overrides)
```

```
!
```

```
// Что перезаписано?
```

```
if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_PURE)  
    SET_BIT(obj->plist[poly].attr,  
        POLY4DV2_ATTR_SHADE_MODE_PURE);
```

```
if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_FLAT)  
    SET_BIT(obj->plist[poly].attr,  
        POLY4DV2_ATTR_SHADE_MODE_FLAT);
```

```
if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_GOURAUD)  
{  
    SET_BIT(obj->plist[poly].attr,  
        POLY4DV2_ATTR_SHADE_MODE_GOURAUD);
```

```
    // Скоро понадобится информация о нормалях!
```

```
    SET_BIT(obj->vlist_local[obj->plist[poly]  
        .vert[0]].attr,  
        VERTEX4DTV1_ATTR_NORMAL);
```

```
    SET_BIT(obj->vlist_local[obj->plist[poly]  
        .vert[1]].attr,  
        VERTEX4DTV1_ATTR_NORMAL);
```

```
    SET_BIT(obj->vlist_local[obj->plist[poly]  
        .vert[2]].attr,  
        VERTEX4DTV1_ATTR_NORMAL);
```

```
} // if
```

```
if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_TEXTURE)  
    SET_BIT(obj->plist[poly].attr,  
        POLY4DV2_ATTR_SHADE_MODE_TEXTURE);
```

```
} // if
```

Заметим, что для реализации затенения по Гуро необходимо установить флаг `VERTEX4DTV1_ATTR_NORMAL`. Это правильно, поскольку нам нужна информация о нормали. В качестве примера приведем фрагмент кода из одной демонстрационной программы. В этом фрагменте показано, как можно загрузить объект и перезаписать при этом информацию об освещении с тем, чтобы *осуществить* затенение по Гуро.

```
Load_OBJECT4DV2_3DSASC(&obj_player,"tie01.asc",  
    &vscale, &vpos, &vrot,  
    VERTEX_FLAGS_OVERRIDE_GOURAUD);
```

## Обновление загрузчика файлов в формате Caligari .COB

Последним по порядку (но не по важности) из представленных будет загрузчик объектов в формате `.COB`. Взглянув на код этой функции, можно увидеть, что она занимает большой объем. Однако основные изменения направлены на поддержку новой структу-

ры OBJECT4DV2. Кроме того, файлы в формате .COB предоставляют возможность использовать текстуру и ее координаты, поэтому в этой функции реализована полная поддержка упомянутых особенностей.

Перед тем как углубиться в суть вопроса, рассмотрим, как происходит обработка файлов в формате .COB. Сначала с помощью значения `plane` поля `color` обозначаем, каким образом затеняется объект, — в чистых цветах или с применением текстуры. Затем с помощью поля `reflectance` выбирается режим затенения — *излучательное*, плоское или затенение по Гуро. Интересно, что при использовании текстур необходимость информации о модели освещения не отпадает. В данной версии игрового процессора для текстур поддерживается излучательное и плоское затенение; интерполяция, осуществляемая при затенении по Гуро, слишком сильно замедляет процесс наложения текстуры. Но это означает, что когда вы создаете модель в формате .COB, вы должны установить цвет многоугольника, для которого задается текстурное отображение (с помощью строки "color: texture map"), а также вид *отражения* — излучательное или плоское затенение (т.е. если поле "Shader class:" имеет значение "reflectance", то в расположенном ниже поле "Shader name:" должно быть значение "constant" или "matte").

**ВНИМАНИЕ**

В нашем процессоре объекты могут иметь только одну текстуру! Эта текстура должна быть размером  $m \times m$ , где  $m$  — это степень двойки, не превышающая 256.

Об этом ограничении не следует забывать, однако это вовсе не означает, что нельзя наложить на объекты текстуру так, чтобы все они хорошо выглядели. Для этого просто понадобится достаточно большая оболочка, скажем,  $128 \times 128$  или  $256 \times 256$ . Затем составляется текстурная карта объекта и с помощью координат текстуры она отображается на объект. В предыдущей главе на рис. 8.52 изображена неплохая оболочка объекта из игры *Quake II*. Из этого рисунка видно, что текстурная карта состоит из множества небольших кусочков, и все, что нужно, — воспользоваться правильными координатами текстуры.

Итак, загрузчик файлов в формате .COB обладает почти теми же функциональными возможностями, что и его предыдущая версия. Однако у него есть и новые возможности, среди которых — умение загружать текстуры и работать с ними. Информация о текстуре считывается из файла в формате .COB. Для куба эти данные имеют следующий вид.

```
Texture Vertices 6
0.000000 0.000000
0.000000 1.000000
0.000000 0.000000
0.000000 1.000000
1.000000 0.000000
1.000000 1.000000
```

При считывании списка поверхностей каждая вторая компонента вершин, задающих многоугольник, является координатой текстуры.

```
Faces 12
Face verts 3 flags 0 mat 4
<0,0> <1,1> <3,5>
Face verts 3 flags 0 mat 4
<0,0> <3,5> <2,4>
Face verts 3 flags 0 mat 0
<0,1> <2,5> <5,4>
Face verts 3 flags 0 mat 0
<0,1> <5,4> <4,0>
Face verts 3 flags 0 mat 1
```

```

<2,2> <3,3> <6,5>
Face verts 3 flags 0 mat 1
<2,2> <6,5> <5,4>
Face verts 3 flags 0 mat 0
<1,0> <7,1> <6,5>
Face verts 3 flags 0 mat 0
<1,0> <6,5> <3,4>
Face verts 3 flags 0 mat 2
<4,4> <5,0> <6,1>
Face verts 3 flags 0 mat 2
<4,4> <6,1> <7,5>
Face verts 3 flags 0 mat 3
<0,4> <4,2> <7,3>

Face verts 3 flags 0 mat 3 face #11
<0,4> <7,3> <1,5>

```

Например, последний многоугольник, **входящий** в каркас куба (поверхность 11), состоит из трех вершин (0,7,1), которым соответствуют вершины текстуры (4,3,5). Этим индексам вершин текстуры соответствуют **следующие** координаты.

```

((1.000000, 0.000000),
 (0.000000, 1.000000),
 (1.000000, 1.000000))

```

Каждая из пар **приведена** в формате (u,v), или в терминах **проекций** на оси x и y текстурной карты. Более подробно отображение текстуры будет рассмотрено далее, в разделе, специально посвященном этому вопросу. Однако я надеюсь, что вы в общих чертах понимаете механику этого процесса. Итак, координаты текстуры загружены. Отлично, однако нужно еще загрузить саму карту текстуры. При этом могут возникнуть определенные проблемы, но если мы позаботимся обо всех деталях, все будет хорошо. Первое правило — для всех объектов можно загружать только одну текстуру. Это означает, что при определении текстурной карты, соответствующей тому или иному материалу, загружается только первая текстурная карта, а остальные (если они существуют) во внимание не принимаются. Ниже приведен фрагмент кода, взятый из загрузчика файлов в формате .SOB, который осуществляет загрузку текстуры.

```

// В структуру OBJECT4DV2 можно загружать все материалы, но
// только одну текстуру. Загрузите первую карту текстуры и
// задайте флаги, не позволяющие задавать для этого объекта
// другие текстуры
if (!obj->texture)
{
    // Этап 1: выделение памяти для битового образа
    obj->texture =
        (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));

    // Загрузка текстуры; используем только конечное имя
    // файла и абсолютный путь к глобальной текстуре
    char filename[80];
    char path_filename[80];
    // Извлечение имени файла
    Extract_Filename_From_Path(
        materials[material_index+ num_materials]

```

```

        .texture_file, filename);

// Составление имени файла с абсолютным путем к нему
strcpy(path_filename, texture_path);
strcat(path_filename, filename);

// Теперь в буфере содержится имя файла, содержащего
// текстуру, и путь к нему. Загрузка битового образа
// (поддерживается 8- и 16-битовый режим.)
Load_Bitmap_File(&bitmap16bit, path_filename);

// Создание битового образа надлежащего размера
// и глубины
Create_Bitmap(obj->texture, 0, 0,
              bitmap16bit.bitmapinfoheader.biWidth,
              bitmap16bit.bitmapinfoheader.biHeight,
              bitmap16bit.bitmapinfoheader.biBitCount);

// Загрузка битового изображения (позже сделаем так,
// чтобы это производилось в 8- и 16-битовом режиме)
if (obj->texture->bpp == 16)
    Load_Image_Bitmap16(obj->texture,
                        &bitmap16bit, 0, 0, BITMAP_EXTRACT_MODE_ABS);
else
    i
    Load_Image_Bitmap(obj->texture, &bitmap16bit, 0, 0,
                      BITMAP_EXTRACT_MODE_ABS);
} // else 8 bit

// Все готово; выгружаем битовый образ
Unload_Bitmap_File(&bitmap16bit);

// Устанавливаем для объекта флаг,
// соответствующий наличию текстуры
SET_BIT(obj->attr, OBJECT4DV2_ATTR_TEXTURES);

} // if

```

Приведенный выше фрагмент кода играет важную **роль**. Он выполняет определенную подготовительную работу, над которой стоит поразмыслить. Фрагмент расположен после разделов кода, загружающих все многоугольники и **ставящих** в соответствие им материалы. Напомним, что в формате **.SOB** это соответствие реализуется с помощью индекса материала, который задается для каждого многоугольника. Однако **информация** о материалах загружается только *после* того, как будет загружен весь **каркас**. Таким образом, следует подождать, пока завершится загрузка каркаса, загрузить материалы, а затем разрешить ссылки на материалы. Мы пока что используем не сами материалы, а только содержащуюся в них текстуру. В любом случае, при каждой загрузке материала, для которого определена текстура, запускается приведенный выше код.

Первый этап, который **осуществляется** в этом коде, состоит в том, чтобы определить, была ли загружена текстура. Если да — то ничего не происходит. В противном **случае** передается управление блоку кода. Сначала с помощью новой функции `Extract_Filename_From_Path()`, код которой приведен ниже, извлекается имя файла, содержащего карту текстуры.

```

char *Extract_Filename_From_Path(char *filepath,
                                  char *filename)
{
    // Эта функция извлекает имя файла из полного имени
    // (с указанием пути к файлу) "../folder/./filename.ext"
    // Функция последовательно просматривает в обратном порядке
    // полное имя в поисках символов "\" или "/", а затем
    // копирует имя файла, начиная с этого места и до конца
    // Далее проверяется корректность извлеченного имени
    if (!filepath || strlen(filepath)==0)
        return(NULL);

    int index_end = strlen(filepath)-1;

    // Поиск имени файла
    while( (filepath[index_end]!='\\') &&
           (filepath[index_end]!='/') &&
           (filepath[index_end] > 0) )
        index_end--;

    // Копирование имени файла в переменную filename
    memcpy(filename, &filepath[index_end+1],
           strlen(filepath)-index_end);

    // Возврат результата
    return (filename);
} // Extract_Filename_From_Path

```

Необходимость этой функции обусловлена отсутствием простого метода, с помощью которого можно было бы извлечь имя файла из полного имени (**включающего** путь к этому файлу). Функция работает, просматривая символы в обратном порядке, начиная с конца строки, и возвращает строку, которая следует после последнего символа \ или /. Пусть, например, для файла с полным именем "d:\files\atari\object.cob" функция вызвана следующим образом.

```

char filename[80];
Extract_Filename_From_Path("d:\files\atari\object.cob", filename);

```

В результате будет возвращено *имя filename="object.cob"*, а это как раз то, что нужно. Это все хорошо, но как же быть с путем к файлу? Вдруг он пригодится, например, чтобы поместить в этот каталог файл с текстурой? Для этого я решил создать путь к *корневому каталогу с текстурами*. Это означает, что в данном каталоге должны находиться все текстуры. В библиотеке T3DLIB7.CPP путь к корневому каталогу определен следующим образом.

```

char texture_path[80] = "./"; // Путь ко всем текстурам.

```

Другими словами, поиск текстур будет производиться в рабочем каталоге, из которого запущен исполняемый файл. Напомним, что **строка** "." обозначает текущий каталог. Таким образом, на данный момент все аудиовизуальные файлы, прилагаемые к демонстрационным программам, должны находиться в тех же **каталогах**, что и сами программы. Впоследствии при создании собственных игр вы, возможно, захотите создать каталог MEDIA\ (для мультимедийных файлов), а также вспомогательные каталоги TEXTURES\ (для текстур), MODELS\ (для моделей), SOUNDS\ (для звуков) и т.д.

Вторая причина, по которой пришлось выполнить всю эту работу по извлечению имени файла, намного важнее. Если мы вспомним, как в формате **Caligari .COB** задается объект с текстурой, то поймем, что здесь таится проблема. Рассмотрим пример материала с текстурой.

```
Mat1V0.06 Id 18623892 Parent 18629556 Size 00000182
mat# 0
shader: phong facet: auto32
rgb 1,0.952941,0.0235294
alpha 1 ka 0.1 ks 0.1 exp 0 ior 1
texture: 36D:\Source\models\textures\wall01.bmp
offset 0,0 repeats 1,1 flags 2
ShBxV0.03 Id 18623893 Parent 18623892 Size 00000658
Shader class: color
Shader name: "texture map" (caligari texture)
Number of parameters: 7
file name: string "D:\Source\models\textures\wall01.bmp"
S repeat: float 1
T repeat: float 1
S offset: float 0
T offset: float 0
animate: bool 0
filter: bool 0
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "phong" (phong)
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.9
specular factor: float 0.1
exponent: float 3
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
ENDV1.00 Id 0 Parent 0 Size 0
```

Обратите внимание, что для файлов с текстурой задаются абсолютные пути! Это означает, что демонстрационная программа, созданная на одном компьютере, не будет работать на другом компьютере, если все вспомогательные файлы не размещены в таких же каталогах. Поэтому, думаю, что лучше всего использовать имя файла и общий путь к файлам с текстурами.

НА ЗАМЕТКУ

Возможно, в загрузчике **trueSpace** есть настройка, позволяющая отменить использование абсолютных путей, однако, потратив некоторое время на эту проблему, я сдался.

Пора подводить итоги в обсуждении новой функции, предназначенной для считывания файлов в формате **.COB**. После того как получена информация об имени и пути к

файлу с текстурой, загружается сам файл. При этом не происходит ничего необычного. Вызывается функция, загружающая битовый образ, а также создается объект изображения, и в него копируется изображение из файла с текстурой. Конечно же, предварительно определяется битовая глубина (8- или 16-битовая), а потом уже на основе этой информации вызывается соответствующая функция.

Здесь возникает необходимость немного поговорить о битовой глубине. На определенном этапе может возникнуть желание поместить 8- и 16-битовые функции в функции-оболочки более высокого уровня, потому что каждый раз перед вызовом функции выполнять проверку обременительно. Я не стал этого делать, потому что, используя ту или иную битовую глубину, мы и так знаем, что именно мы используем. Эту задачу несложно реализовать с помощью указателей на функции или с помощью виртуальных функций. При этом удастся избежать дополнительного использования ресурсов. В общем, здесь есть над чем подумать...

Следующий вопрос, на котором хотелось бы остановиться, касается функции, предназначенной для перезаписи вершин, которая входит в состав загрузчика файлов в формате .COB. Код модели освещения переписывать бессмысленно (хотя вы можете попытаться), однако если с чем и придется повозиться — так это координаты текстуры. Некоторые программы, в которых создаются модели в формате .COB, могут выводить в файлы информацию о координатах текстур в несколько неудобном виде. Например, координаты *u* и *v* могут быть переставлены, они могут быть представлены в инвертированной системе координат и т.д. Чтобы позволить загрузчику представлять эти данные в удобном виде, в нем введены такие флаги.

```
// Инвертирование координаты текстуры u
#define VERTEX_FLAGS_INVERT_TEXTURE_U 0x0080
// Инвертирование координаты текстуры v
#define VERTEX_FLAGS_INVERT_TEXTURE_V 0x0100
// Перестановка координат текстуры u и v
#define VERTEX_FLAGS_INVERT_SWAP_UV 0x0800
```

С помощью оператора OR их можно объединять в логических выражениях с другими флагами, которые могут понадобиться в формате .COB, такими как VERTEX\_FLAGS\_TRANSFORM\_LOCAL\_WORLD и др.

Если уж речь зашла о координатах текстуры, следует заметить, что остался еще один вопрос, который следовало бы обсудить. В большинстве моделей используются координаты текстуры в интервале от 0 до 1. Другими словами, они нормированы на единицу. Для наших функций растеризации это не совсем удобно. Поэтому после того, как координаты текстуры, наконец, будут записаны в специальный массив-кэш и функция-загрузчик завершит свою работу, необходимо убедиться, что координаты текстуры преобразованы согласно масштабу текстуры. Например, если загружена текстура размерами 64x64 и координаты текстуры составляют (0.2, 0.34), то обе координаты следует умножить на  $(64-1)=63$ :

$(63 \cdot 0.2, 63 \cdot 0.34) = (12.6, 21.42)$ .

НА ЗАМЕТКУ

Обратите внимание, что координаты текстуры так и остались числами с плавающей точкой. Это позволяет сохранять точность со степенью, которая не ограничена размерами пикселей.

Пора подводить итог изучения нового загрузчика файлов в формате .COB. Вызывается эта функция так же, как и предыдущие загрузчики. Ниже приведен пример ее вызова, взятый из одной из последних демонстрационных программ. С помощью этой инструкции загружается модель TIE с текстурой. При этом следует учесть, что координата *v* ин-

вертируется аналогично тому, как это происходит в системе моделирования Caligari trueSpace, в которой используется правосторонняя система координат.

```
Load_OBJECT4DV2_COB(&obj_player,"tie02.cob",
    &vscale,&vpos,&vrot,
    VERTEX_FLAGS_INVERT_TEXTURE_V |
    VERTEX_FLAGS_SWAP_YZ |
    VERTEX_FLAGS_TRANSFORM_LOCAL_WORLD );
```

## Обзор растеризации многоугольников

Перед тем как перейти к изложению затенения по Гуро и отображению текстуры, вернемся еще раз к вопросу о растеризации многоугольников. Этот вопрос уже рассматривался в книге *Программирование игр для Windows. Советы профессионала*; кроме того, я вкратце говорил об этом в предыдущей главе, однако он возникает везде, где используется трехмерная графика. В подавляющем большинстве программ приходится иметь дело с растеризацией, интерполяцией или дискретизацией, поэтому этими вопросами стоит хорошо овладеть.

С учетом сказанного, я переписал основные функции, предназначенные для растеризации, чтобы повысить их точность. Дело в том, что первоначальные версии этих функций работали только с целочисленными координатами. Кроме того, было принято соглашение о заполнении многоугольников, а также реализованы некоторые другие возможности. Обсудим эти изменения, чтобы вы могли лучше их понять.

## Растеризация треугольников

Треугольники можно выводить несколькими способами, включая такой, при котором положение их боковых сторон отслеживается по алгоритму Брезенхама (Bresenham) или с помощью обычной интерполяции. Я предпочитаю интерполяцию, поскольку она более понятна. Рассмотрим еще раз, как все это работает. Все, что нужно сделать, — найти вершины растеризуемой версии треугольника. На рис. 9.9 они обозначены маленькими точками. После того как будет найдено расположение пикселей, из которых состоит каждая строка развертки треугольника, вывод треугольника на экран сведется к обычному заполнению памяти для каждой точки, как показано на рис. 9.10.

Поиск точек, принадлежащих сторонам треугольника, — это, в основном, просто интерполяция, которая выполняется следующим образом.

Известно, что высота треугольника равна

$$d_y = (y_2 - y_0),$$

а длины проекций его боковых сторон на ось  $x$  равны

$$d_{x\_left} = (x_2 - x_0),$$

$$d_{x\_right} = (x_1 - x_0).$$

Наклон правой стороны находится из соотношения

$$m_{right} = d_y / d_{x\_right} = (y_2 - y_0) / (x_1 - x_0).$$

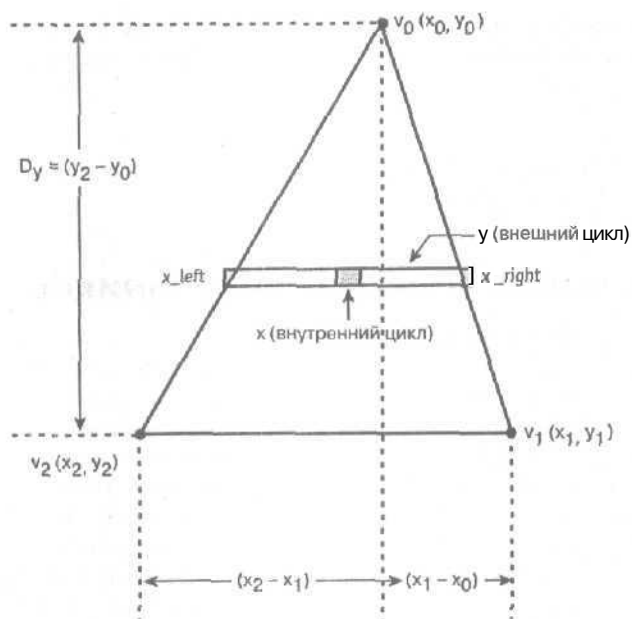


Рис. 9.9. Схема растеризации треугольника



Рис. 9.10. Растеризация одной строки развертки

Однако нам нужен не сам наклон, а обратная к нему величина. Дело в том, что наклон характеризует скорость изменения координаты  $y$  с изменением координаты  $x$ . Другими словами, если мы произведем сдвиг по оси  $x$  на один пиксель, то координата  $y$  точки, находящейся на стороне треугольника, изменится на величину наклона. Вывод треугольника производится построчно. При переходе от одной строки развертки к другой координата  $y$  каждый раз получает постоянное приращение, равное 1. Соответствующее приращение координат  $x$  правой и левой боковых сторон треугольника при  $d_y = 1$  равно

$$d_{x\_left} = (x_2 - x_0) / (y_2 - y_0),$$

$$d_{x\_right} = (x_1 - x_0) / (y_2 - y_0).$$

Вот и все! Это и есть полный алгоритм, предназначенный для вывода треугольника с горизонтальной нижней стороной. Треугольник с горизонтальной верхней стороной строится аналогично, и соответствующий алгоритм вы можете разработать самостоятельно. Рассмотрим приведенный ниже псевдокод, реализующий описанный алгоритм.

```

void Draw_Triangle(float x0, float y0, // Вершина 0
                  float x1, float y1, // Вершина 1
                  float x2, float y2, // Вершина 2
                  int color)          // Цвет
{
    // Эта функция производит растеризацию треугольника
    // с горизонтальной нижней стороной

    // Вычисляем коэффициент интерполяции для левой стороны
    float dx_left = (x2 - x0)/(y2 - y0);

    // Вычисляем коэффициент интерполяции для правой стороны
    float dx_right = (x1 - x0)/(y2 - y0);

    // Начальные условия для интерполяции координат правой
    // и левой сторон
    float x_left = x0;
    float x_right = x0;

    // Вход в цикл растеризации
    for (int y = y0; y <= y1; y++)
    {
        // Вывод строки развертки
        Draw_Line(x_left, x_right, y, color);

        // Прибавление прироста
        x_left += dx_left;
        x_right += dx_right;
    } // for y
} // DRAW_Triangle

```

На этой функции основана функция, предназначенная для растеризации треугольников. Функция `Draw_Triangle()` работает неплохо, однако она не лишена некоторых недостатков. Во-первых, в ней используются числа с плавающей точкой. На самом деле, это не проблема, поскольку на современных компьютерах операции с такими числами выполняются с той же скоростью, что и операции с целыми числами. Проблемы могут возникнуть на этапе растеризации в процессе преобразования целых чисел в числа с плавающей точкой и обратно, поэтому кратко остановимся на этом вопросе.

Реальную проблему представляет функция `Draw_Line()`. Предполагается, что в нее передаются числа с плавающей точкой. Возможно, это можно организовать, однако на определенном этапе эти величины необходимо будет преобразовать в целые числа. Вот *здесь-то* и возникает проблема. Каким образом лучше выполнять это преобразование — путем округления снизу, обычного округления или округления сверху? Перечисленные виды округления выполняются с помощью стандартных функций C/C++ `ceil()` и `floor()`. Все эти вопросы относятся к проблеме, которую называют *соглашением о заполнении* (*fill convention*).

#### СОВЕТ

На рис. 9.11 приведен график, полученный в результате использования математической функции `ceil(x)`. Эта функция возвращает наименьшее целое число, не меньшее  $x$ . Функция `floor(x)` возвращает наибольшее целое число, не превышающее  $x$ . График этой функции приведен на рис. 9.12.

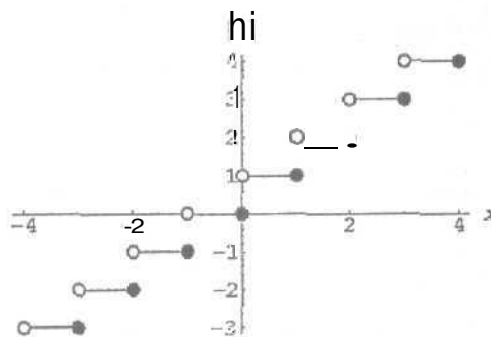


Рис. 9.11. График функции  $\text{ceil}()$

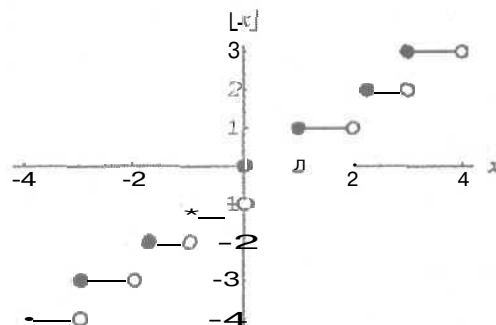


Рис. 9.12. График функции  $\text{floor}()$

Соглашение о заполнении определяет, как производится растеризация многоугольника по отношению к начальному и конечному пикселям в вертикальном (сверху вниз) и горизонтальном (слева направо) направлениях. Скоро мы вернемся к этому важному вопросу, а сейчас продолжим обсуждение функции растеризации и поговорим о некоторых других ее особенностях.

Итак, какие еще нужно сделать предположения относительно приведенного выше прототипа функции, кроме соглашения о заполнении?

1. Ничего не было сказано о треугольниках общего вида, и этот вопрос нужно подробно обсудить.
2. Не делалось никаких замечаний по поводу отсечения, будь то отсечение в пространстве изображения или пространстве объекта. Этот вопрос тоже заслуживает внимания.

Сначала рассмотрим первый вопрос. В этой книге используется функция растеризации треугольников, разработанная в предыдущей книге *Программирование игр для Windows. Советы профессионала*. Напомним, что все треугольники можно разбить на такие типы:

1. треугольники с горизонтальной нижней стороной;
2. треугольники с горизонтальной верхней стороной;
3. треугольники **общего** вида.

Треугольники каждого из перечисленных типов изображены на рис. 9.13. Надеюсь, к этому времени вы уже поняли, что, возможно, быстрее было бы провести предвари-

тельные тесты и не писать более общую функцию растеризации треугольников. Оснежим память, воспроизведем ход рассуждений.



Рис. 9.13. Виды треугольников

Если внимательно рассмотреть процесс вывода треугольника, то можно заметить, что в нем имеется внутренний цикл от  $y_0$  до  $y_1$ . В этом цикле определяются абсциссы точек, лежащих на правой и левой сторонах треугольника, а затем эти точки соединяются линией. В треугольнике общего вида в некоторой точке наклон правой или левой боковой стороны изменится. Это означает, что вместо того, чтобы разбивать треугольник на два треугольника с горизонтальной нижней и верхней сторонами, в цикл от  $y_0$  до  $y_1$  можно поместить код проверки, определяющий, в какой точке произойдет изменение наклона.

Если мы воспользуемся разбиением треугольника общего вида на два специальных треугольника, то в функции растеризации нужно будет применять две вспомогательных функции для растеризации каждого из специальных треугольников. Если же мы возьмем на вооружение подход, при котором возможно изменение наклона, то нам понадобится код проверки, подобный приведенному ниже. Предполагается, что производится растеризация треугольника, изображенного на рис. 9.14.

```
for (y=y0; y<y2; y++)
{
    Draw_Line(x_left,x_right,y.color);

    x_left +=dx_left;
    x_right+=dx_right;

    // Проверяем, изменился ли наклон
    if (y==y1) dx_right = new_dx_right;
} // for
```

По сути, во внутреннем цикле нужно проверять, когда интерполяция дойдет до новой стороны треугольника, и соответствующим образом изменить в этот момент величину наклона. Однако при этом возникает один тонкий вопрос. Вне зависимости от того, как производится вывод треугольника, — путем его разбиения на два составляющих треугольника или с помощью проверки изменения наклона, — в месте изменения наклона возникает возможность допустить ошибку. Чтобы понять, в чем тут проблема, ознакомьтесь с нижней частью рис. 9.14. На нем изображено место стыка двух сторон. Сначала мы выводим пиксели верхней стороны, доходим до последнего пикселя и меняем величину наклона, чтобы приступить к выводу следующей стороны. Однако откуда следует начать этот вывод — с текущего пикселя или с того, который расположен ниже? Здесь следует

проявить осторожность, чтобы не перезаписать пиксель (на самом деле, если даже это случится, то ничего страшного не произойдет). Хуже, если произойдет неправильный сдвиг координаты  $y$ . Чтобы избежать этого, перед переходом к следующему шагу следует извлечь точные значения координаты вершины  $(x_1, y_1)$ . Повторим все сначала. Когда текущая координата  $y$  становится равной ординате вершины треугольника, координату  $x$  необходимо сдвигать не на величину нового наклона  $\text{new\_dx\_right}$ , а так, чтобы она стала равной фактической координате  $x_1$  вершины.

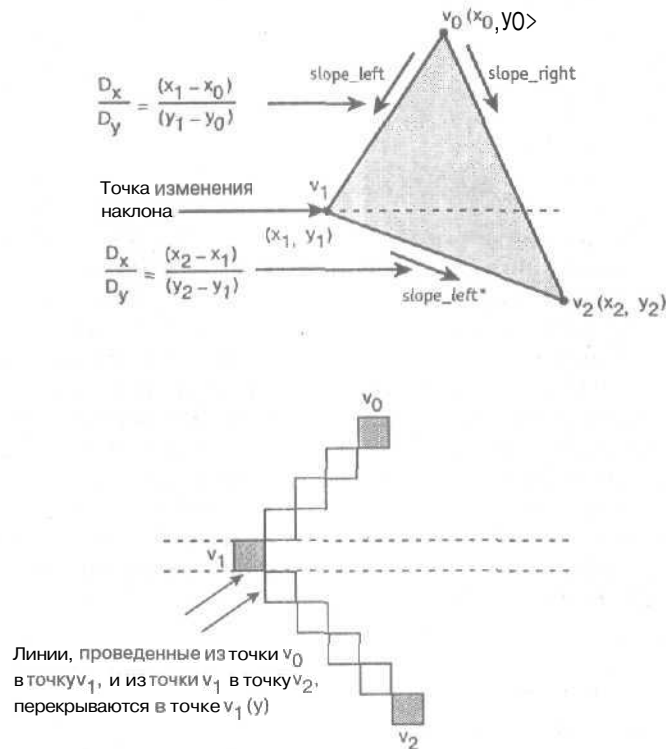


Рис. 9.14. Учет изменения наклона в процессе растеризации треугольника

## Соглашение о заполнении

В процессе растеризации многоугольников в трехмерном игровом процессоре обычно преследуется цель воспроизвести изображение некоторого непрозрачного объекта. В каркасе объекта часто встречаются многоугольники с общими вершинами. Хотелось бы выводить смежные многоугольники таким образом, чтобы они не перекрывались вдоль общих сторон. Чтобы понять, о чем идет речь, взгляните на рис. 9.15а. На нем изображены два треугольника, из которых состоит ромбообразный четырехугольник. Верхний треугольник оканчивается в строке 20 и здесь же начинается нижний! Поэтому нам нужно либо прекратить вывод верхнего треугольника в строке 19, либо начать вывод нижнего треугольника в строке 21. В качестве второго примера рассмотрим рис. 9.15б. На нем изображены два треугольника с общей вертикальной стороной, тоже образующие ромбообразный четырехугольник. Как и в предыдущем случае, возникает вопрос о расположении границы раздела. Он заключается в том, выводить ли самые правые пиксели

левого треугольника в столбце 49 или начинать вывод самых левых пикселей правого треугольника в столбце 51? Ответы на эти вопросы и составляют соглашение о заполнении.

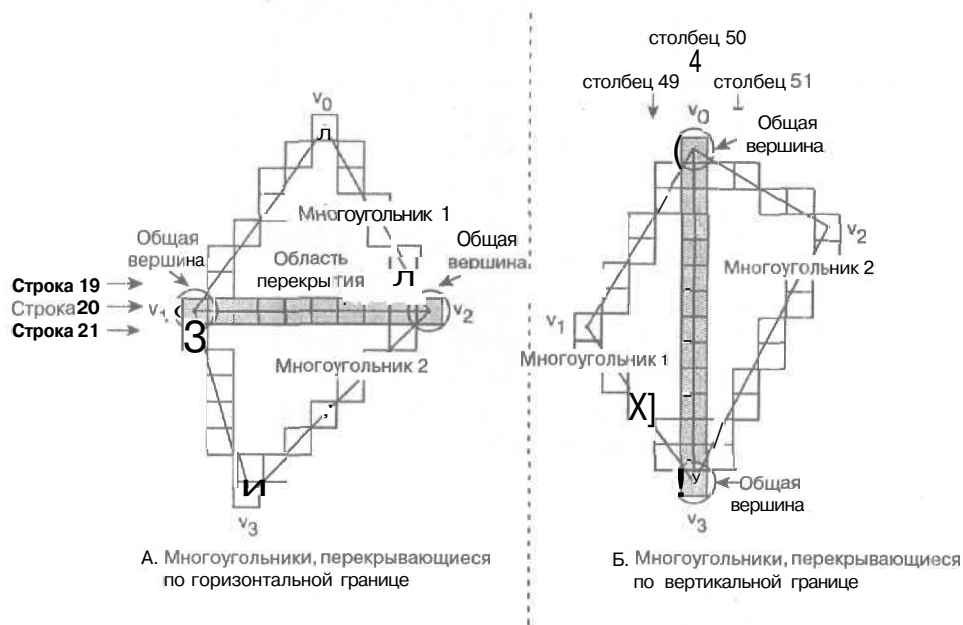


Рис. 9.15. Треугольники с общими вершинами и сторонами

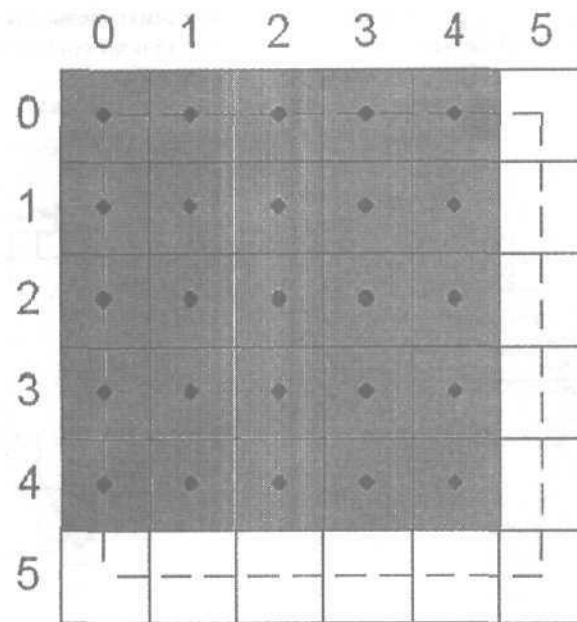
Мы будем использовать соглашение, известное как *соглашение о заполнении верхних левых пикселей* (top-leftfill convention). Такое же соглашение для растеризации принято в Direct3D. Это означает, что в процессе визуализации квадрата со стороной, равной 5 пикселям, и началом в точке (0,0) не заполняются строка и столбец с номером 5 (рис. 9.16).

Как и следовало ожидать, изображенный квадрат занимает 25 пикселей. Его ширина определяется как разность номеров правого и левого столбцов, а высота — как разность номеров верхнего и нижнего столбцов. Однако последняя строка и последний столбец не заполняются.

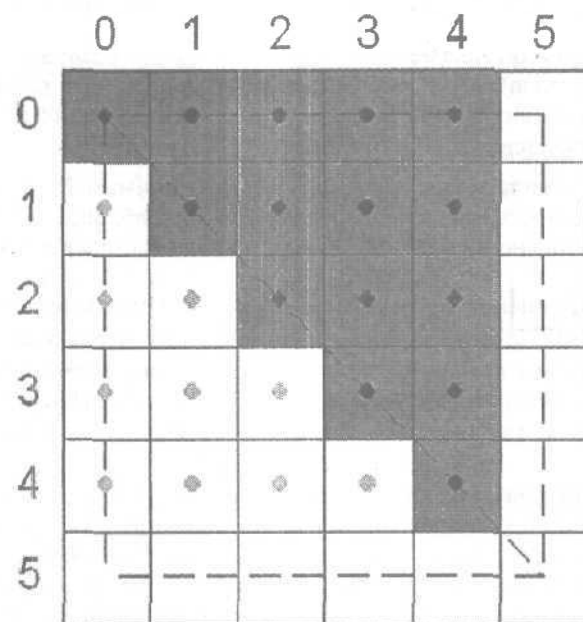
Соглашение о заполнении верхних и левых пикселей определяет, какие действия следует выполнить, если сторона треугольника проходит через какой-нибудь пиксель. На рис. 9.17 изображены два треугольника: один — с вершинами в точках (0,0), (5,0) и (5,5), а другой — с вершинами в точках (0,5), (0,0) и (5,5). Первый треугольник занимает площадь, равную 15 пикселям (обозначенным черным цветом), а второй — площадь, равную всего 10 пикселям (обозначенным серым цветом). Причина в том, что общая граница расположена слева по отношению к первому треугольнику.

Это соглашение мы и хотим реализовать, поскольку, как я уже упоминал, оно является стандартом. Для этого просто нужно правильно преобразовать координаты пикселя в целочисленные значения с помощью функции `ceil()`. Если, например, производится растеризация треугольника с горизонтальной нижней стороной в интервале ординат от  $y_0$  до  $y_1$ , то эти координаты преобразуются следующим образом:

```
ystart = ceil(y0)
yend = ceil(y1)-1
```



*Рис. 9.16. Соглашение о заполнении верхних и левых пикселей применительно к квадрату*



*Рис. 9.17. Соглашение о заполнении применительно к смежным треугольникам*

Во внутреннем цикле, **соответствующем** изменению координаты  $x$  в каждой строке развертки, также понадобится выполнить преобразование к целочисленным значениям:

```
xs = ceil(x_left)
xe = ceil(x_right)-1
```

Итак, подведем итог. Сначала нужно вычислить функцию  $\text{ceil}()$  от начального и конечного параметров цикла по  $y$  и то же самое — от начального и конечного параметров цикла по  $x$ , а потом выводить строку **развертки**.

Все просто, но ничего ли не упущено? Поскольку перед началом растеризации теряется информация о расположении пикселей, это нужно учесть перед определением сдвига по  $x$ . Это означает, что на самом деле координата  $y$  изменяется в пределах от  $y_0$  до  $y_1$ , но, в соответствии с принятым нами правилом заполнения верхних и левых индексов, начальное и конечное положение равны  $\text{ceil}(y_0)$  и  $\text{ceil}(y_1)-1$ , соответственно. Это следует учесть, определяя значения  $x_{\text{left}}$  и  $x_{\text{right}}$  (если выводится треугольник с горизонтальной нижней стороной).

Рассмотрим рис. 9.18. На нем подробно проиллюстрировано побочное действие, возникающее в результате приоритетного заполнения верхних и левых пикселей, на левый интерполянт. Здесь важно убедиться в том, что изменено положение  $y_{\text{start}}$ . Это легко делается с помощью выражения

$x_{\text{left}} - x_{\text{left}} + (y_{\text{start}} - y_0) * dx_{\text{left}}$

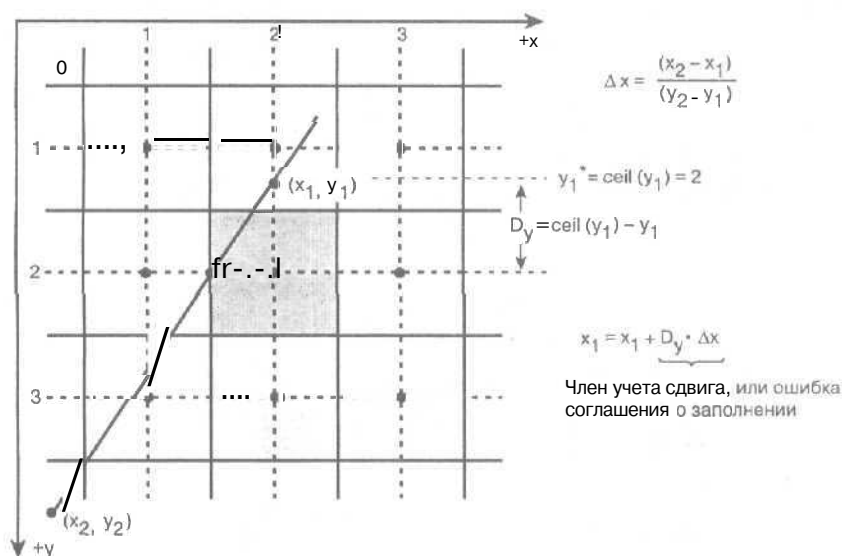


Рис. 9.18. Учет обрезания координаты и надлежащее вычисление сдвига

Другими словами, нужно ввести небольшую поправку для  $x$ , возникающую в результате того, что значение  $y_0$  становится значением  $\text{ceil}(y_0)$ . Таким образом, мы вычисляем разность между исходным значением  $y_0$  и значением  $\text{ceil}(y_0)$ , а затем умножаем его на величину, обратную наклону левого края. Таким образом будет получено правильное значение начального положения координаты  $x$ , соответствующее левому **интерполируемому** значению. Аналогичный процесс должен быть выполнен и для правой стороны.

Вот и все. Единственным дополнительным приготовлением, которое следует выполнить перед растеризацией треугольника, является предварительная сортировка его вершин. Порядок сортировки — сверху вниз и слева направо.

## Отсечение графического изображения

Перед тем как перейти к работе с реальными треугольниками, поговорим о процедуре отсечения по границам области. Эта тема уже несколько раз обсуждалась, но на этот раз она будет рассматриваться в связи с растеризацией треугольников. Можно было бы обрезать все треугольники, ориентируясь на прямоугольную область экрана (рис. 9.19). Однако оказалось, что такой способ требует больше возни, чем он того заслуживает, и что отсечение лучше выполнять в процессе растеризации. Добиться этого довольно просто. Необходимо выяснить, отсечение какого вида нам нужно — отсечение по оси  $y$ , по оси  $x$  или оно не производится, после чего выбирается блок кода, выполняющий эту процедуру, как показано на рис. 9.20. Конечно же, самый простой случай — это когда отсечение не требуется. За ним следует случай, когда выполняется отсечение по оси  $y$ . При этом отсекается часть треугольника, выступающая за рамки экрана сверху или снизу (а если треугольник большой, то у него могут отсекаться обе выступающие части).

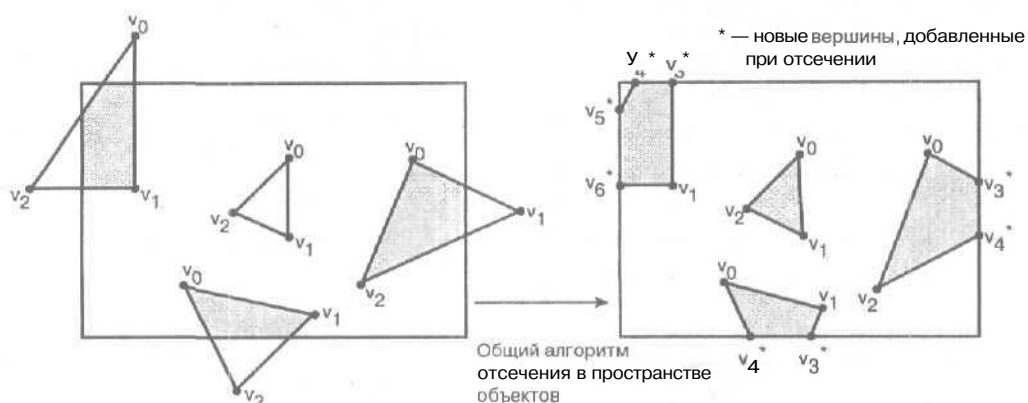


Рис. 9.19. Отсечение прямоугольной областью экрана

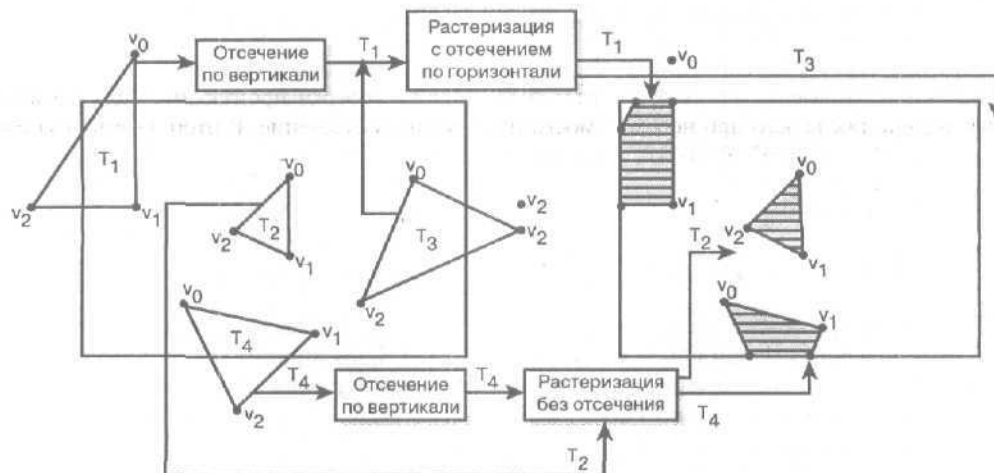


Рис. 9.20. Частные случаи отсечения, основанные на геометрии

Например, чтобы выполнить отсечение по верхней границе экрана (в большинстве случаев ей соответствует координата  $y = 0$ ), ордината верхней вершины треугольника сравнивается с нулем. Если эта ордината отрицательная, переменной  $y$  присваивается значение 0. Однако здесь нужно соблюдать осторожность, правильно вычислив соответствующие абсциссы точек, лежащих на правой и левой сторонах треугольника при новом значении  $y$ . Еще раз обратим ваше внимание на то, что это предварительная фаза, и чтобы не потерять точность, следует быть осторожным (рис. 9.20).

Низ треугольника отсекает намного проще, чем верх. Для этого достаточно прекратить растеризацию, дойдя до последней строки развертки экрана. При этом не нужно вычислять координаты  $x$  левого и правого пикселей, ограничивающих строку развертки треугольника, потому что, продвигаясь сверху вниз, мы всегда заранее знаем, когда следует остановиться.

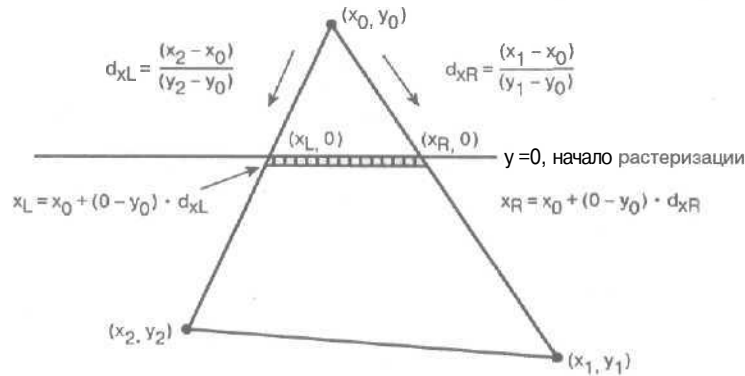


Рис. 9.21. Правильное вычисление пределов растеризации по оси  $x$  после обрезания по оси  $y$

Последний и самый сложный случай — это когда треугольник, обрабатываемый процедурой растеризации, выходит за рамки правого или (и) левого края окна. Отсечение по левому краю аналогично отсечению по верхней границе экрана. Чтобы начать формирование усеченной строки развертки с нужного места, достаточно просто присвоить переменной, соответствующей координате  $x$ , значение 0. Затем аналогичная проверка производится для правого края экрана, после чего при необходимости производится отсечение. В итоге результаты сохраняются и строка развертки выводится на экран — как видим, ничего сложного.

#### СОВЕТ

Немного позже мы реализуем трехмерное отсечение, однако оказывается, что не стоит выполнять полное трехмерное отсечение по границам области видимости, поскольку в результате треугольник может оказаться шестиугольником. Однако никак нельзя обойтись без отсечения по  $z$ -плоскости, проходящей в непосредственной близости от (предполагаемого) объектива камеры. В результате некоторые треугольники могут превратиться в четырехугольники, которые затем снова понадобится разбивать на треугольники. Однако эти операции можно производить на уровне списка визуализации, где они не приведут к значительной потере производительности.

## Новые функции, связанные с обработкой треугольников

Теперь, когда мы ознакомились с некоторыми подробностями, касающимися растеризации треугольников, рассмотрим новые функции, предназначенные для их визуализации. В этих функциях принимаются во внимание некоторые из изложенных выше

концепций. Пришлось переписать все функции, производящие визуализацию треугольников, которые работают как в 8-битовом, так и в 16-битовом режимах. Основное беспокойство вызывало то, что в старых версиях не поддерживались координаты, выражаемые числами с плавающей точкой. Поэтому в новых функциях такая поддержка реализована. Кроме того, введены флаги условной компиляции, позволяющие выбирать точность, с которой работает функция растеризации. Дело в том, что чем выше точность, тем медленнее работают функции. Я решил, что в данном случае условная компиляция подойдет лучше, чем оптимизация с помощью ассемблера. Будут приведены листинги не всех функций, а лишь нескольких, в которых ярче всего проявляются новые функциональные возможности. Рассмотрим функцию, выполняющую разбиение треугольников. В этой функции вызывается 16-битовая версия функции низкого уровня, производящей вывод треугольников с горизонтальной нижней или верхней стороной.

```
void Draw_Triangle_2D2_16(float x1, float y1,
                          float x2, float y2,
                          float x3, float y3,
                          int color,
                          UCHAR *dest_buffer, int mempitch)
{
    // Функция выводит треугольник в буфер назначения.
    // Она разбивает все треугольники на пары треугольников
    // с нижней и верхней горизонтальными сторонами

    float temp_x, // Используется для сортировки
    temp_y,
    new_x;

#ifdef DEBUG_ON
    // Отслеживает состояние визуализации
    debug_polys_rendered_per_frame++;
#endif

    // Проверка наличия горизонтальных и вертикальных сторон
    if ((FCMP(x1,x2) && FCMP(x2,x3)) ||
        (FCMP(y1,y2) && FCMP(y2,y3)))
        return;

    // Сортировка вершин
    if (y2 < y1)
    {
        SWAP(x1,x2,temp_x);
        SWAP(y1,y2,temp_y);
    } // if

    // Теперь вершины p1 и p2 следуют в правильном порядке
    if (y3 < y1)
    {
        SWAP(x1,x3,temp_x);
        SWAP(y1,y3,temp_y);
    } // if

    // Сравниваем значения y3 и y2
```

```

if (y3 < y2)
{
    SWAP(x2,x3,temp_x);
    SWAP(y2,y3,temp_y);
} // if

//Тривиальные проверки необходимости отсечения
if (y3 < min_clip_y || y1 > max_clip_y ||
    (x1 < min_clip_x && x2 < min_clip_x && x3 < min_clip_x) |
    (x1 > max_clip_x && x2 > max_clip_x && x3 > max_clip_x))
    return; *

// Проверяем, горизонтальна ли верхняя сторона
// треугольника
if (FCMP(y1,y2))
{
    Draw_Top_Tri2_16(x1,y1,x2,y2,x3,y3,color,
        dest_buffer,mempitch);
} // if
else
    if (FCMP(y2,y3))
    {
        Draw_Bottom_Tri2_16(x1,y1,x2,y2,x3,y3,color,
            dest_buffer,mempitch);
    } // если низ горизонтальный
else
    {
        // Треугольник общего вида, который нужно
        // разбить на два
        new_x = x1 + (y2-y1)*(x3-x1)/(y3-y1);

        // Вывод каждого из составляющих треугольников
        Draw_Bottom_Tri2_16(x1,y1,new_x,y2,x2,y2,color,
            dest_buffer,mempitch);
        Draw_Top_Tri2_16(x2,y2,new_x,y2,x3,y3,color,
            dest_buffer,mempitch);
    } // else
} // Draw_Triangle_2D2_16

```

Прототип этой функции совпадает с прототипом ее предыдущей версии, Draw\_Triangle\_2D\_16(). Единственное различие в том, что все вычисления выполняются над числами с плавающей точкой, и что координаты пикселей тоже передаются в этом формате. Далее мы рассмотрим функцию, которая в 16-битовом режиме выводит треугольники с горизонтальной нижней стороной.

```

void Draw_Bottom_Tri2_16(float x1, float y1,
    float x2, float y2,
    float x3, float y3,
    int color,
    UCHAR *_dest_buffer, int mempitch)
{
    // Функция выводит треугольники
    // с горизонтальной нижней стороной

```

```

float dx_right, // Отношение dx/du для правой стороны
dx_left, // Отношение dx/du для левой стороны
xs, xe, // Начальная и конечная абсциссы краев
height // Высота треугольника
temp_x, // Временные переменные, используемые
temp_y, // при сортировке
right // Используются при отсечении
left;

int iy1, iy3, loop_y;

// Преобразование указателя на буфер назначения
// к типу ushort
USHORT *dest_buffer = (USHORT *)_dest_buffer;

// Адрес назначения следующей строки развертки
USHORT *dest_addr = NULL;

// Повторное вычисление шага памяти, выраженное
// в 16-битовых словах
mempitch = (mempitch >> 1);

// Проверяем порядок следования x1 и x2
if (x3 < x2)
{
    SWAP(x2, x3, temp_x);
} // if swap

// Вычисляем наклоны
height = y3 - y1;

dx_left = (x2 - x1) / height;
dx_right = (x3 - x1) / height;

// Задаем начальные точки
xs = x1;
xe = x1;

ftff (RASTERIZER_MODE == RASTERIZER_ACCURATE)
// Выполняем отсечение по оси y
if (y1 < min_clip_y)
{
    // Вычисляем новые значения переменных xs и ys
    xs = xs + dx_left * (-y1 + min_clip_y);
    xe = xe + dx_right * (-y1 + min_clip_y);

    // Сброс значения y1
    y1 = min_clip_y;

    // Убеждаемся, что выполняется соглашение о
    // заполнении верхних и левых пикселей
    iy1 = y1;

```

```

} // if (выход за верхнюю границу экрана)
else
{
    // Убеждаемся, что выполняется соглашение о
    // заполнении верхних и левых пикселей
    iy1 = ceil(y1);

    // Задаем правильные значения переменных xs и xe
    xs = xs + dx_left * (iy1 - y1);
    xe = xe + dx_right * (iy1 - y1);
} // else

if (y3 > max_clip_y)
{
    // Отсечение по оси y
    y3 = max_clip_y;

    // Убеждаемся, что выполняется соглашение о
    // заполнении верхних и левых пикселей
    iy3 = y3 - 1;
} // if
else
{
    // Убеждаемся, что выполняется соглашение о
    // заполнении верхних и левых пикселей
    iy3 = ceil(y3) - 1;
} // else
#endif

#if (RASTERIZER_MODE == RASTERIZER_FAST) ||
(RASTERIZER_MODE == RASTERIZER_FASTEST)
// Отсечение по оси y
if (y1 < min_clip_y)
{
    // Задаем новые значения переменных xs и ys
    xs = xs + dx_left * (-y1 + min_clip_y);
    xe = xe + dx_right * (-y1 + min_clip_y);

    // Сброс значения y1
    y1 = min_clip_y;
} // if (треугольник выходит за верхнюю границу экрана)

if (y3 > max_clip_y)
    y3 = max_clip_y;

// Убеждаемся, что выполняется соглашение о заполнении
// верхних и левых пикселей
iy1 = ceil(y1);
iy3 = ceil(y3) - 1;
#endif

// Вычисление начального адреса в видеопамяти
dest_addr = dest_buffer + iy1 * mipmapitch;

```

```

// Проверяем, нужно ли выполнять отсечение по оси x
if (x1 >= min_clip_x && x1 <= max_clip_x &&
    x2 >= min_clip_x && x2 <= max_clip_x &&
    x3 >= min_clip_x && x3 <= max_clip_x)
{
    // Вывод треугольника
    for (loop_y = iy1; loop_y <= iy3; loop_y++,
        dest_addr+=mempitch)
    {
        // Вывод линии
        Mem_Set_WORD(dest_addr+(unsigned int)(xs),color,
            (unsigned int)((int)xe-(int)xs+1));

        // Устанавливаем начальную и конечную точки
        xs+=dx_left;
        xe+=dx_right;
    } // for
} // if (отсекать по оси x не нужно)
else
{
    // Медленная версия отсечения по оси x

    // Вывод треугольника
    for (loop_y = iy1; loop_y <= iy3;
        loop_y++,dest_addr+=mempitch)
    {
        // Отсечение по оси x
        Left = xs;
        right = xe;

        // Задаем начальную и конечную точки
        xs+=dx_left;
        xe+=dx_right;

        // Отсечение стороны
        if (left < min_clip_x)
        {
            left = min_clip_x;

            if (right < min_clip_x)
                continue;
        }

        if (right > max_clip_x)
        {
            right = max_clip_x;

            if (Left > max_clip_x)
                continue;
        }
    }
    // Вывод линии
}

```

```

        Mem_Set_WORD(dest_addr+(unsigned int)(left),
            color,
            (unsigned int)((int)right-(int)left+1));
    } // for "
} // else (нужно отсечение по оси x)
} // Draw_Bottom_Tri2_16

```

В имени этой функции, как и в имени предыдущей, согласно принятому соглашению об именах, добавлено число 2, а ее интерфейс остается тем же. Ознакомившись с приведенным выше листингом, обратите внимание на то, как реализована поддержка соглашения о заполнении, а также на флаги условной компиляции, выделенные полужирным шрифтом. Приведем определение этих флагов.

```

// Определение флагов условной компиляции в новых функциях
// растеризации. Эти флаги используются для изменения
// логики, чтобы избежать наличия 80 миллионов различных
// функций. Скорее всего, трех флагов будет достаточно
#define RASTERIZER_ACCURATE 0 // Наивысшая точность и
// соблюдение соглашения о заполнении

#define RASTERIZER_FAST 1
#define RASTERIZER_FASTEST 2

// Задаем режим, который будет использоваться
// в игровом процессоре
#define RASTERIZER_MODE RASTERIZER_ACCURATE

```

Чтобы принять другие условия компиляции, достаточно изменить выделенную строку в файле `T3DLIB7.H`, заменив, например, флаг `RASTERIZER_ACCURATE` флагом `RASTERIZER_FAST`. При этом растеризация выполняется быстрее, но это достигается за счет некоторой потери точности. В этой функции реализуются почти все возможности, которые обсуждались выше. Функция-аналог, производящая вывод треугольников с горизонтальной верхней стороной, работает аналогично. Кроме того, как видно из приведенного ранее списка функций, существуют версии, различающиеся битовыми режимами.

Теперь, когда вам уже надоела растеризация, немного поговорим о некоторой оптимизации, которую можно провести в функциях растеризации.

## Оптимизация

Замечательная особенность функций растеризации — то, что они получились такими короткими. Особо отметим, что оптимизируемая область очень компактна. Итак, где же можно провести оптимизацию? Применяемые в функциях алгоритмы вполне надежны, поэтому в них не стоит вносить кардинальные изменения — растеризация есть растеризация. Кроме того, во внутренних циклах очень мало кода. В основном, по паре строк с операциями сложения и вызовами функций, производящих вывод линий. Единственное, что можно ускорить, — это повнимательнее присмотреться к преобразованию данных и выявить математические операции, в которых нет необходимости. Рассмотрим приведенный ниже фрагмент кода, представляющий собой внутренний цикл функции растеризации.

```

// Вывод треугольника
for (loop_y = iy1; loop_y <= iy3; loop_y++,
    dest_addr += mipmapitch)
{
    // Вывод линии
}

```

```
Mem_Set_WORD(dest_addr+(unsigned int)(xs), color,
              (unsigned int)((int)xe-(int)xs+1));
```

```
// Задаем начальную и конечную точки
xs+=dx_left;
xe+=dx_right;
} // for
```

Единственная часть, замедляющая работу функции, — это выделенная строка, в которой вызывается функция `Mem_Set_WORD()`. Это встраиваемая функция, однако вызов, о котором идет речь, все же не внушает доверия. Чтобы увеличить производительность функции, можно было бы непосредственно применить в ней встроенный ассемблер. Что еще вызывает беспокойство — так это выполняемое преобразование типов, которое тоже может стать причиной замедления. По сути, здесь предпринята попытка реализовать соглашение о заполнении горизонтальных строк, но это достигается лишь частично, поскольку не вызывается функция `ceil()` для конечных точек всех строк развертки. Это приводит к тому, что строки развертки получаются длиннее, чем следовало бы. Однако такая неточность окупается уменьшением количества вызовов функции `ceil()`, которые могли бы существенно замедлить работу функции. Для строгого соблюдения соглашения о заполнении понадобится более производительная версия функции `ceil()`, чтобы ее можно было использовать все время и анализировать, как выполняется преобразование к типу `int`. Дело в том, что в подобных фрагментах напрашивается применение ассемблера, поскольку компилятор генерирует недостаточно эффективный код для внутреннего цикла. Однако основная оптимизация с помощью ассемблера откладывается до того времени, пока мы не перейдем к заключительной части книги.

Поговорим немного о функциях `ceil()` и `floor()`. Они входят в состав стандартных библиотек C/C++, и даже если они вызываются только в точных версиях функций, производящих растеризацию треугольников, то все равно это 10–20 тысяч возможных вызовов, поэтому оптимизация вполне оправдана. Один из способов повысить производительность функции `ceil()` — самостоятельно написать ее версию для того диапазона чисел, в котором предполагается производить округление. Другими словами, заранее известно, что будут выводиться треугольники, вершины которых выражаются координатами в диапазоне `screen_width×screen_height ± некоторое отклонение`. Тогда оказывается возможным организовать таблицу соответствия для функций `ceil()` и `floor()` в этом диапазоне. Например, рассмотрим такой алгоритм вычисления функции `ceil()`:

```
ceil(x)=
{ if x >= 0: ceil(x) = (x == int(x)) ? x : int(x) + 1
{ if x < 0 : ceil(x) = int(x)
```

Решающим фактором при вычислении функции `ceil(x)` является наличие десятичной части у ее аргумента. Сначала проверяем, есть ли у числа `x` дробная часть. Если она есть и число положительно, то его целая часть увеличивается на единицу; в противном случае аргумент остается без изменений. Если же число `x` отрицательно, его дробная часть просто отбрасывается. Ниже приведена реализация описанного алгоритма.

```
inline int Ceiling(float x)
{
    if (x < 0) return (int)x;
    else if ((int)x < x) return ((int)x+1);
    return x;
} // if
```

На встроенном ассемблере это можно выполнить с **помощью** нескольких команд. Аналогично можно оптимизировать функцию `floor()`. Однако основная цель данного обсуждения — обратить ваше внимание на такой простой факт: не имеет **значения**, с какой скоростью работает процессор при выполнении математических операций с плавающей точкой или выполняются операции с целыми числами, если для **вычисления** конечного местонахождения пикселя приходится тратить много времени на преобразования типов. Вот почему имеет смысл иногда применять числа с фиксированной точкой. При этом все математические вычисления можно выполнять с **числами** одного и того же типа, а на этапе растеризации — **использовать** старшие 16 битов переменных, в которых **хранятся** координаты **пикселей**. Такой подход оправдан во многих отношениях, и он был использован как для реализации затенения по Гуро, так и для аффинного отображения текстуры.

Наконец, заметим, что основное внимание в плане оптимизации следует уделять внутренним циклам. Как мы убедились, подпрограммы, с помощью которых ведется обработка многоугольников, могут вызываться около 10–20 тысяч раз для каждого каркаса. Поэтому все, что можно "подчистить" на этапе задания параметров многоугольника, стоит **того**. Эти рассуждения еще раз подтверждают, что, возможно, лучше будет не разбивать треугольники общего вида на два треугольника с горизонтальными **нижней** и **верхней** сторонами, а выполнить его растеризацию как целого. Это позволит два вызова функций для ряда треугольников заменить одним.

Я уже воспользовался этой стратегией, разрабатывая функции, выполняющие затенение по Гуро и аффинное отображение текстуры. В этих функциях треугольник общего вида обрабатывается как единая фигура, без разбиения на части.

Это лишь повод для размышлений. **Существует** много способов выполнить **растеризацию** треугольников, и всегда найдется возможность ускорить эту процедуру. Я думаю, что с помощью ассемблера, **многопотчности** и других средств можно увеличить скорость работы предложенных здесь функций примерно от 2 до 10 раз.

## Реализация затенения по Гуро

Наконец мы дошли до сути. Ведь основная задача этой главы — рассказать о затенении по Гуро и об аффинном отображении текстуры, но до сих пор нам приходилось заниматься подготовительными операциями! Рассмотрим рис. 9.22, на котором изображен снимок экрана, полученный в результате работы игрового процессора. Обратите внимание на цветные огоньки текстуры и на гладкое затенение многоугольников. **Выглядит** здорово, правда? А теперь посмотрим, как это все работает.

Как уже говорилось в предыдущей главе, затенение и освещение по Гуро отличается от плоского затенения. Вместо **того** чтобы вычислять всего одну нормаль для каждого многоугольника и одну интенсивность освещения для всей **его** поверхности, в алгоритме затенения по Гуро приходится вычислять интенсивность **освещения** в каждой вершине. На этапе растеризации интенсивности освещения в вершинах интерполируются по всему треугольнику.

На рис. 9.23 проиллюстрирован **процесс** интерполяции. По сути, процессор освещения обрабатывает каждую нормаль, проведенную в вершине многоугольника, вместо **того**, чтобы производить расчеты с единой нормалью для всего многоугольника. Затем вычисляется RGB-значение, соответствующее интенсивности падающего **света**, которое сохраняется в структуре вершины как ее цвет. Далее эти цвета линейно интерполируются по всему треугольнику, в результате чего получаем треугольник с затенением по Гуро!

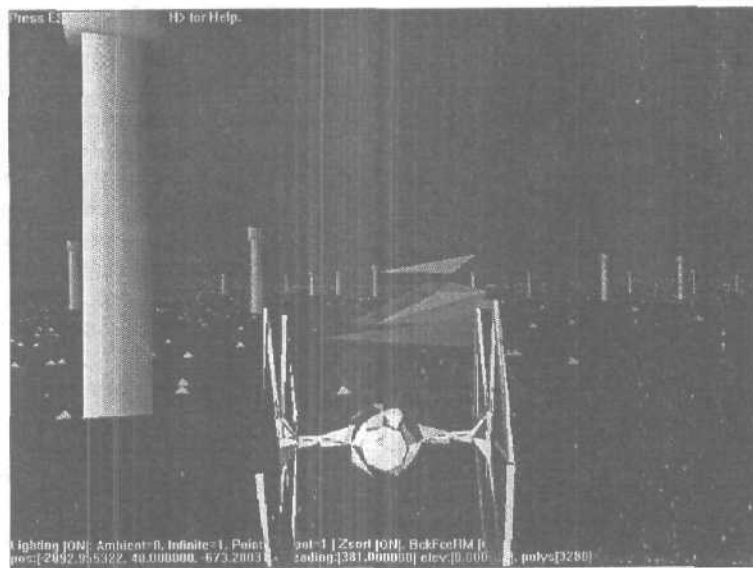


Рис. 9.22. Трехмерный игровой процессор с затенением по Гуро и отображением текстуры в действии

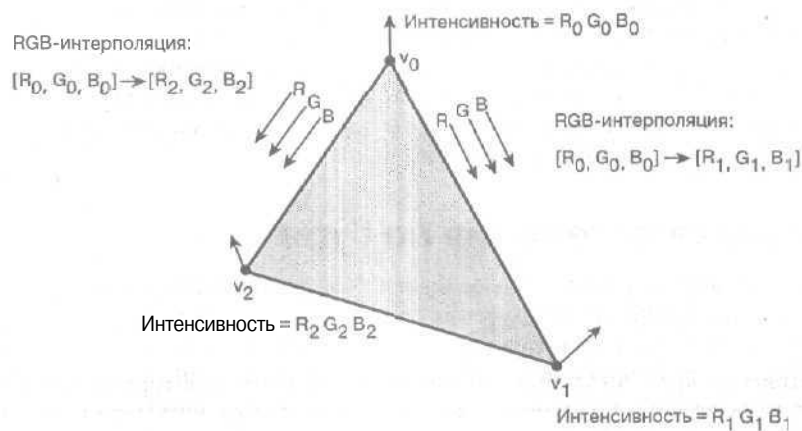


Рис. 9.23. Интерполяция по треугольнику с затенением по Гуро

В процессе разработки функции, производящей затенение по Гуро, необходимо решить две задачи.

1. Часть I. Следует создать процессор освещения, способный определять интенсивность освещения на основе данных о нормали к поверхности. Однако такой процессор уже есть. Осталось только сделать так, чтобы для каждого многоугольника с затенением по Гуро вычислялась интенсивность освещения на основе нормалей, проведенных во всех его вершинах.
2. Часть II. Необходимо разработать программу, осуществляющую растеризацию по Гуро. Эта программа должна быть способна обрабатывать цвета, хранящиеся в структуре каждой вершины многоугольника, и выполнять интерполяцию в промежуточных областях между вершинами.

Первая задача уже почти выполнена, поэтому предположим на минуту, что на вход модуля освещения подан объект `OBJECT4DV2` или список визуализации `RENDERLIST4DV2`. В результате на выходе получаем многоугольники с такими видами затенения: излучательное, плоское и затенение по Гуро. Если многоугольник затенен по Гуро, цвета его вершин хранятся в массиве `lit_color[0..2]`. В противном случае, то есть для многоугольников с излучательным и плоским затенением, элемент массива `lit_color[0]` будет содержать цвет всего многоугольника. Таким образом, если предположить, что мы придерживаемся этой технологии, проблема становится совсем простой. Нам нужна такая функция, осуществляющая растеризацию треугольника, чтобы интерполяция цвета пикселей в ней выполнялась не только вдоль сторон треугольника, но и вдоль строк развертки.

## Затенение по Гуро без освещения

Цвета можно задавать вручную, вычислять с помощью процессора освещения, основываясь на параметрах материала, извлекать из структуры, хранящей параметры статического места действия или как-то еще. Важно то, что в процессе визуализации треугольника производится интерполяция цветов вершин по всей области треугольника. Это очень простая задача, с которой мы уже много раз сталкивались.

На рис. 9.24 проиллюстрирована процедура, которую нужно реализовать. У нас есть треугольник общего вида с вершинами  $v_0, v_1, v_2$  в том порядке, в котором они перечислены. Кроме того, пусть в каждой вершине задан RGB-цвет в формате 8.8.8. Что происходит в процессе растеризации треугольника? Сначала вычисляются величины, обратные наклонам левой и правой сторон, и полученные значения присваиваются переменным `dx_left` и `dx_right`. Эти величины характеризуют, насколько быстро изменяется абсцисса точки, лежащей на стороне треугольника, с изменением ее ординаты. Таким образом, при переходе от одной строки развертки к другой абсцисса начальной и конечной точек строки развертки изменяется на величины `dx_left` и `dx_right` соответственно, после чего строка выводится на экран. Именно это нам и нужно для затенения по Гуро; в ходе этого затенения необходимо выполнять интерполяцию цветов вдоль сторон треугольника по оси  $y$ . Попытаемся реализовать это с помощью псевдокода в упрощенной версии. В RGB-версии все будет происходить точно так же, понадобится лишь трехкратный повтор, по одной интерполяции для каждого канала. На данном этапе не хочется беспокоиться по поводу соглашения о заполнении, сортировке вершин и других подобных вещах. Сейчас мы займемся именно интерполяцией.

Рассмотрим процесс интерполяции подробнее. В конечных версиях функций будут использованы переменные с именами, которые несколько отличаются от тех, что мы используем сейчас. На рис. 9.24 изображен треугольник общего вида, в котором вершины  $v_0, v_1$  и  $v_2$  пронумерованы в порядке их обхода по часовой стрелке. Кроме того, на этом рисунке приведены обозначения обратных наклонов и координат вершин. Рассмотрим интерполяцию треугольника шаг за шагом.

1. Начальный этап — задаем начальные значения для левой и правой сторон.

С помощью этих величин отслеживаются положения точек интерполяции, лежащих на левой и правой сторонах треугольника:

```
xl_edge = x0
xr_edge = x0
```

Приведенные ниже величины служат для отслеживания интерполируемой интенсивности на левой и правой сторонах треугольника:

```
ilEdge = i0
irEdge = i0
```

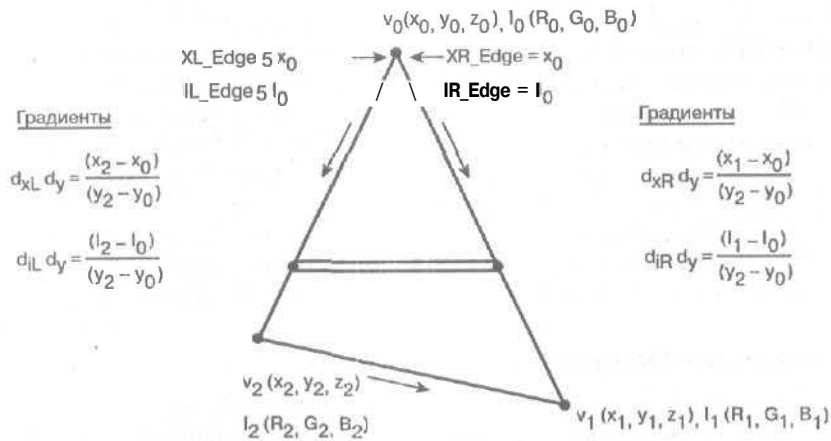


Рис. 9.24. Схема интерполяции при затенении по Гуро

2. Вычисляем величины, обратные наклонам левой и правой сторон.

Обратный наклон левой стороны определяет, как изменяется абсцисса точки, лежащей на этой стороне, с изменением ординаты. Обратный наклон вычисляется по формуле:

$$dxLdy = (x2 - x0) / (y2 - y0)$$

Аналогично вычисляем обратный наклон правой стороны:

$$dxRdy = (x1 - x0) / (y1 - y0)$$

Далее вычисляем градиенты интенсивности, которые определяют, как меняется **ИНТЕНСИВНОСТЬ** освещения с изменением ординаты точки:

$$diLdy = (I2 - I0) / (y2 - y0)$$

Аналогично вычисляем соответствующую величину для правой стороны:

$$diRdy = (I1 - I0) / (y1 - y0)$$

3. Организовываем цикл по у от у0 до у2 (первая половина треугольника).
4. Этап горизонтальной растеризации. На данном этапе происходит вывод строки развертки. Однако в строке нужно выводить по одному пикселю за раз, поскольку интерполяция производится как по вертикали, так и по горизонтали.

Задаем начальные значения для х и интенсивности:

$$i = iL\_edge$$

$$x = xL\_edge$$

Вычисляем горизонтальный градиент интенсивности:

$$dix = (iR\_edge - iL\_edge) / (xR\_edge - xL\_edge)$$

5. Производим растеризацию строки развертки; выводим по одному пикселю за раз с учетом интерполяции:

for (x = xL\_edge to xR\_edge)

begin

setpixel(x,y,i)

// Обновляем значение интенсивности

i = i + dix

end

6. Обновляем значения интерполируемых по вертикали величин и переходим к следующему циклу.

Сначала обновляем положения левого и правого краев:

```
xl_edge = xl_edge + dxldy  
xr_edge = xr_edge + dxrdy
```

Теперь обновляем значения интенсивности:

```
il_edge = il_edge + dildy  
ir_edge = ir_edge + dirdy
```

Переходим к следующему шагу по у.

7. Выполняем все то же для нижней части треугольника, однако перед этим обновляем значение обратного наклона для левого края, что соответствует переходу к конечной вершине v1.

Потратим немного времени, чтобы пересмотреть реализованный в виде псевдокода алгоритм и убедиться, что принцип его работы вполне понятен. По сути, растеризация, которую мы собираемся реализовать, идентична обычной растеризации за исключением того, что мы отслеживаем значения интенсивностей вдоль левой и правой сторон треугольника. При выводе каждой строки, вместо того, чтобы вывести все входящие в ее состав пиксели одним цветом, необходимо интерполировать интенсивность вдоль строки от левого до правого ее края. А теперь попробуем преобразовать сформулированный выше псевдокод в грубый код на C/C++, чтобы представить весь алгоритм в целом.

```
Draw_Gouraud_Triangle(  
    float xO, float yO, float iO, //Вершина O  
    float x1, float y1, float i1, //Вершина 1  
    float x2, float y2, float i2 //Вершина 2  
)  
{  
    // Предположим, что вершины расположены так, как  
    // показано на рисунке  
  
    // Начальные положения  
    xl_edge = xO;  
    xr_edge = xO;  
  
    // Начальные интенсивности  
    il_edge = iO;  
    ir_edge = iO;  
  
    // Начальные значения обратных наклонов сторон  
    // и градиентов интенсивности по вертикали  
    dxldy = (x2 - xO)/(y2 - yO);  
    dxrdy = (x1 - xO)/(y1 - yO);  
    dildy = (i2 - iO)/(y2 - yO);  
    dirdy = (i1 - iO)/(y1 - yO);  
  
    // Вход в цикл по у (только для первой  
    // половины треугольника)  
    for (y=yO; y < y2; y++)  
    {  
        // Задаем начальные значения для х и интенсивности
```

```

i = iLedge;
x = xl_edge;

// Вычисляем горизонтальный градиент интенсивности i
dix = (ir_edge - iL_edge) / (xr_edge - xl_edge);

// Выполняем растеризацию строки развертки пиксель
// за пикселем с учетом интерполяции

for (x = xl_edge; x < xr_edge; x++)
{
    // Вывод пикселя с координатами (x,y)
    // и интенсивностью i
    setpixel(x,y,i);

    // Обновляем значение интенсивности
    i = i + dix;
} // for

// Обновляем положения строки развертки и значения
// интенсивностей на ее краях
xl_edge = xl_edge + dxldy;
xr_edge = xr_edge + dyrdy;
iL_edge = iL_edge + dildy;
ir_edge = ir_edge + dirdy;

} // for y

// Здесь нужно добавить код для второй половины
// треугольника. Для этого следует обновить значение
// обратного наклона левой стороны, а затем выполнить
// те же действия

```

```

} // Draw_Gouraud_Triangle

```

Подведем итог. Визуализация треугольника с затенением по **Гуро**, или растеризация треугольника, при которой интенсивности или значения цветов каждой вершины **плавно** интерполируются во внутренней области по вертикали и горизонтали, — это именно то, что реализовано в псевдокоде и приведенной выше минимальной функции на **C/C++**. Что же нужно для того, чтобы реально воплотить все это? По правде говоря, достаточно от интерполяции единых значений "интенсивности" перейти к интерполяции значений красного, зеленого и синего цветов. Значение цвета в формате **RGB** интерполируется по вертикали вдоль левой и правой сторон треугольника, а также по горизонтали вдоль строк развертки (рис. 9.25).

Единственное, чего нужно добиться в процессе этой интерполяции — скорость. Интерполяция вдоль обеих сторон производится в шести цветовых каналах и двух каналах растеризации. Кроме того, при выводе строки развертки необходимо интерполировать значение цвета в формате **RGB** от левого до правого края строки, а это **еще** три канала (см. рис. 9.25). Все это должно работать с хорошей скоростью!

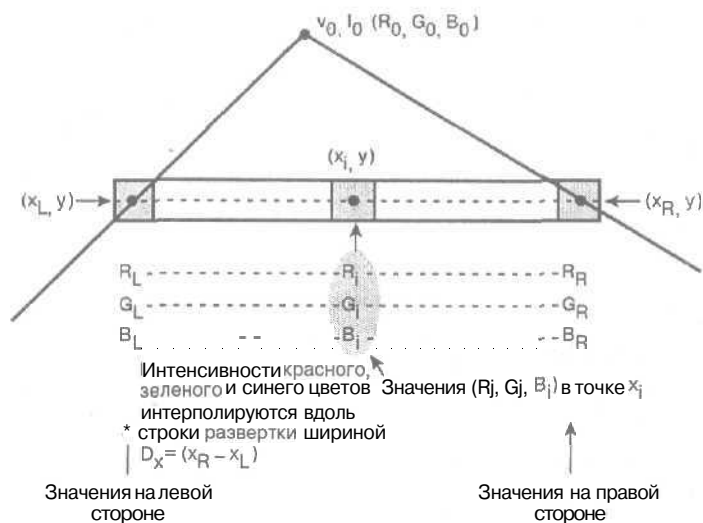


Рис. 9.25. Растеризация строки развертки при полноцветном затенении по Гуро

Учитывая последнее замечание, мне пару раз пришлось переписывать код. В итоге я пришел к варианту, в котором используются числа с фиксированной точкой в формате 16.16. Этот вариант предпочтительнее не потому, что операции над числами с фиксированной точкой выполняются быстрее, чем над числами с плавающей точкой. Просто я хотел избежать преобразования чисел с плавающей точкой в целые числа, которое понадобилось бы при выводе пикселей и вычислении конечных значений слов, хранящих цвета в формате RGB. Дело в том, что для подобных операций компилятор C/C++ генерирует неэффективный код, а использовать ассемблер мне не хотелось. Я решил привести листинг этой функции, несмотря на ее значительный объем. Это единственный листинг функции в данной главе, который включен в текст из-за того, что функция довольно большая и сложная. Тем не менее, функция заслуживает того, чтобы привести ее код, поскольку он создан в манере RISC, что позволяет извлечь выгоды из конвейерного режима процессора Pentium. Я старался программировать в стиле, позволяющем достичь максимальной степени распараллеливания. Листинг приводится только для треугольника первого вида (т.е. с горизонтальной нижней стороной), однако он содержит все необходимые для ознакомления элементы.

```
void Draw_Gouraud_Triangle16(
    POLYF4DV2_PTR face,    // Указатель на поверхность
    UCHAR*_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch)         // Число байтов в строке - 320, 640
                          // и т.п.
{
    // Эта функция выводит треугольник, затененный по Гуро,
    // на основе аффинного отображения текстуры. Вместо
    // того, чтобы интерполировать координаты текстуры, мы
    // просто интерполируем значения (R,G,B) по области
    // треугольников. Поскольку интерполируются другие
    // величины, переменные имеют такие значения:
    // u->красный, v->зеленый, w->синий
}
```

```

int v0    =0,
    v1    = 1,
    v2    = 2,
    temp  = 0,
    tri_type = TRI_TYPE_NONE,
    irestart = INTERP_LHS;

int dx,dy,dxl,dyr, // Общие поправки
    u,v,w,
    du,dv,dw,
    xi,yi, // Текущие интерполируемые значения x,y
    ui,vi,wi, // Текущие интерполируемые значения u,v
    index_x,index_y, // Параметры цикла.
    x,y, // Здесь хранятся общие значения x,y
    xstart, xend, ystart, yrestart, yend, xl,
    dxdxl, xr, dxdyr, dudyl, ul, dvdy, vl, dwdy,
    wl, dudyr, ur, dvdyr, vr, dwdyr, wr;

int x0,y0,tu0,tv0,tw0, // Кэширование параметров вершин
    x1,y1,tu1,tv1,tw1, .
    x2,y2,tu2,tv2,tw2;

intr_base0, g_base0, b_base0, // Базовые цвета
    r_base1, g_base1, b_base1,
    r_base2, g_base2, b_base2;

USHORT *screen_ptr = NULL,
    *screen_line = NULL,
    *textmap = NULL,
    *dest_buffer=(USHORT *)_dest_buffer;

#ifdef DEBUG_ON
    // Отслеживание состояния визуализации
    debug_polys_rendered_per_frame++;
#endif

// Приводим шаг памяти в соответствие с длиной слова,
// т.е. делим его на 2
mem_pitch >>=1;

// Первая тривиальная проверка необходимости отсечения
if (((face->tvlist[0].y < min_clip_y) &&
    (face->tvlist[1].y < min_clip_y) &&
    (face->tvlist[2].y < min_clip_y)) ||

    ((face->tvlist[0].y > max_clip_y) &&
    (face->tvlist[1].y > max_clip_y) &&
    (face->tvlist[2].y > max_clip_y)) ||

    ((face->tvlist[0].x < min_clip_x) &&
    (face->tvlist[1].x < min_clip_x) &&
    (face->tvlist[2].x < min_clip_x)) ||

    ((face->tvlist[0].x > max_clip_x) &&
    (face->tvlist[1].x > max_clip_x) &&
    (face->tvlist[2].x > max_clip_x)) &&

```

```

    (face->tvlist[2].x > max_clip_x)))
return;

// Треугольник специального вида
if ( ((face->tvlist[0].x==face->tvlist[1].x) &&
      (face->tvlist[1].x==face->tvlist[2].x)) ||
      ((face->tvlist[0].y==face->tvlist[1].y) &&
      (face->tvlist[1].y==face->tvlist[2].y)))
    return;

// Сортировка вершин
if (face->tvlist[v1].y < face->tvlist[v0].y)
{SWAP(v0,v1,temp);}

if (face->tvlist[v2].y < face->tvlist[v0].y)
{SWAP(v0,v2,temp);}

if (face->tvlist[v2].y < face->tvlist[v1].y)
{SWAP(v1,v2,temp);}

// Проверяем наличие у треугольника горизонтальной
// стороны
if (face->tvlist[v0].y==face->tvlist[v1].y)
{
    // Задаем тип треугольника
    tri_type = TRI_TYPE_FLAT_TOP;

    // Сортируем вершины слева направо
    if (face->tvlist[v1].x < face->tvlist[v0].x)
    {SWAP(v0,v1,temp);}
} // if
else
    // Проверяем наличие у треугольника горизонтальной
    // стороны
    if (face->tvlist[v1].y==face->tvlist[v2].y)
    {
        // Задаем тип треугольника
        tri_type = TRI_TYPE_FLAT_BOTTOM;
        // Сортируем вершины слева направо
        if (face->tvlist[v2].x < face->tvlist[v1].x)
        {SWAP(v1,v2,temp);}
    } // if
    else
    {
        // Треугольник общего вида
        tri_type = TRI_TYPE_GENERAL;
    } // else

// Предполагается, что цвет задан в формате 5.6.5.
// Для поддержки форматов 5.6.5 и 5.5.5 необходимо
// создавать две версии, поскольку мы не можем позволить
// себе вызов функции во внутреннем цикле
_RGB565FROM16BIT(face->lit_color[v0],
                  &r_base0, &g_base0, &b_base0);

```

```

_RGB565FROM16BIT(face->lit_color[v1],
    &r_base1, &g_base1, &b_base1);
_RGB565FROM16BIT(face->lit_color[v2],
    &r_base2, &g_base2, &b_base2);
// Преобразуем в 8-битовый формат
r_base0 <=< 3; g_base0 <=< 2; b_base0 <=< 3;
// Преобразуем в 8-битовый формат
r_base1 <=< 3; g_base1 <=< 2; b_base1 <=< 3;
// Преобразуем в 8-битовый формат
r_base2 <=< 3; g_base2 <=< 2; b_base2 <=< 3;

// Извлекаем вершины для обработки;
// сейчас они упорядочены
x0 = (int)(face->tvlist[v0].x+0.5);
y0 = (int)(face->tvlist[v0].y+0.5);

tu0 = r_base0; tv0 = g_base0; tw0 = b_base0;

x1 = (int)(face->tvlist[v1].x+0.5);
y1 = (int)(face->tvlist[v1].y+0.5);

tu1 = r_base1; tv1 = g_base1; tw1 = b_base1;

x2 = (int)(face->tvlist[v2].x+0.5);
y2 = (int)(face->tvlist[v2].y+0.5);

tu2 = r_base2; tv2 = g_base2; tw2 = b_base2;

// Задаем значение новой интерполяции
yrestart = y1;

// Определяем тип треугольника
if (tri_type & TRI_TYPE_FLAT_MASK)
(
    if (tri_type == TRI_TYPE_FLAT_TOP)
    {
        // Вычисляем все параметры
        dy = (y2 - y0);
        dxdyl = ((x2 - x0) << FIXP16_SHIFT)/dy;
        dudyl = ((tu2 - tu0) << FIXP16_SHIFT)/dy;
        dvdy1 = ((tv2 - tv0) << FIXP16_SHIFT)/dy;
        dwdyl = ((tw2 - tw0) << FIXP16_SHIFT)/dy;
        dxdyr = ((x2 - x1) << FIXP16_SHIFT)/dy;
        dudyr = ((tu2 - tu1) << FIXP16_SHIFT)/dy;
        dvdyr = ((tv2 - tv1) << FIXP16_SHIFT)/dy;
        dwdyr = ((tw2 - tw1) << FIXP16_SHIFT)/dy;

        // Проверяем необходимость отсечения по y
        if (y0 < min_clip_y)
        {
            // Определяем величину выступа за край
            dy = (min_clip_y - y0);
            // Вычисляем новые начальные значения для
            // левой стороны
            x1 = dxdyl*dy + (x0 << FIXP16_SHIFT);

```

```

    ul = dxdyl*dy + (tu0 << FIXP16_SHIFT);
    vl = dvdyl*dy + (tv0 << FIXP16_SHIFT);
    wl = dwdyl*dy + (tw0 << FIXP16_SHIFT);
    // Вычисляем новые начальные значения для
    // правой стороны
    xr = dxdyr*dy + (x1 << FIXP16_SHIFT);
    ur = dudyr*dy + (tu1 << FIXP16_SHIFT);
    vr = dvdyr*dy + (tv1 << FIXP16_SHIFT);
    wr = dwdyr*dy + (tw1 << FIXP16_SHIFT);
    // Вычисляем новое начальное значение y
    ystart = min_clip_y;
} // if
else
{
    // Отсечение не нужно
    // Задаем начальные величины
    xl = (x0 << FIXP16_SHIFT);
    xr = (x1 << FIXP16_SHIFT);
    ul = (tu0 << FIXP16_SHIFT);
    vl = (tv0 << FIXP16_SHIFT);
    wl = (tw0 << FIXP16_SHIFT);
    ur = (tu1 << FIXP16_SHIFT);
    vr = (tv1 << FIXP16_SHIFT);
    wr = (tw1 << FIXP16_SHIFT);
    // Задаем начальное значение y
    ystart = y0;
} // else

} // if (верхняя сторона горизонтальная)
else
{
    // Нижняя сторона должна быть горизонтальной.
    // Вычисляем все параметры
    dy = (y1 - y0);
    dxdyl = ((x1 - x0) << FIXP16_SHIFT)/dy;
    dudyl = ((tu1 - tu0) << FIXP16_SHIFT)/dy;
    dvdyl = ((tv1 - tv0) << FIXP16_SHIFT)/dy;
    dwdyl = ((tw1 - tw0) << FIXP16_SHIFT)/dy;
    dxdyr = ((x2 - x0) << FIXP16_SHIFT)/dy;
    dudyr = ((tu2 - tu0) << FIXP16_SHIFT)/dy;
    dvdyr = ((tv2 - tv0) << FIXP16_SHIFT)/dy;
    dwdyr = ((tw2 - tw0) << FIXP16_SHIFT)/dy;

    // Проверяем, нужно ли отсекать по y
    if (y0 < min_clip_y)
    {
        // Вычисляем величину выступа
        dy = (min_clip_y - y0);
        // Вычисляем новые начальные значения для
        // левой стороны
        xl = dxdyl*dy + (x0 << FIXP16_SHIFT);
        ul = dudyl*dy + (tu0 << FIXP16_SHIFT);
        vl = dvdyl*dy + (tv0 << FIXP16_SHIFT);
    }
}

```

```

wl = dwdyl*dy + (two << FIXP16_SHIFT);
// Вычисляем новые начальные значения для
// правой стороны
xr = dxdyr*dy + (x0 << FIXP16_SHIFT);
ur = dudyr*dy + (tu0 << FIXP16_SHIFT);
vr = dvdyr*dy + (tv0 << FIXP16_SHIFT);
wr = dwdyr*dy + (tw0 << FIXP16_SHIFT);
// Вычисляем новое начальное значение y
ystart = min_clip_y;
} // if
else
{
    // Отсекать не нужно.
    // Задаем начальные значения
    xl = (x0 << FIXP16_SHIFT);
    xr = (x0 << FIXP16_SHIFT);
    ul = (tu0 << FIXP16_SHIFT);
    vl = (tv0 << FIXP16_SHIFT);
    wl = (tw0 << FIXP16_SHIFT);
    ur = (tu0 << FIXP16_SHIFT);
    vr = (tv0 << FIXP16_SHIFT);
    wr = (tw0 << FIXP16_SHIFT);
    // Задаем начальное значение y
    ystart = y0;
} // else
} // else (нижняя сторона горизонтальна)
// Проверяем, нужно ли отсекай внизу (всегда)
if ((yend - y2) > max_clip_y)
    yend = max_clip_y;

// Проверяем, необходимо ли отсечение по
// горизонтали
if ((x0 < min_clip_x) || (x0 > max_clip_x) ||
    (xl < min_clip_x) || (xl > max_clip_x) ||
    (x2 < min_clip_x) || (x2 > max_clip_x))
{
    // Версия с отсечением
    // Устанавливаем указатель на точку экрана
    // в начальную строку
    screen_ptr = dest_buffer + (ystart * mem_pitch);
    for (yi = ystart; yi <= yend; yi++)
    {
        // Вычисляем положение конечных точек
        xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
        xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
        // Вычисляем начальные значения
        // интерполируемых величин u,v,w
        ui = ul + FIXP16_ROUND_UP;
        vi = vl + FIXP16_ROUND_UP;
        wi = wl + FIXP16_ROUND_UP;
        // Вычисляем интерполируемые u,v
        if ((dx = (xend - xstart)) > 0)
        {

```

```

    du = (ur - ul)/dx;
    dv = (vr - vl)/dx;
    dw = (wr - wl)/dx;
} // if
else
{
    du = (ur - ul);
    dv = (vr - vl);
    dw = (wr - wl);
} // else

////////////////////////////////////

// Проверяем, нужно ли отсекать по x левую
// часть строки
if (xstart < min_clip_x)
{
    // Вычисляем перекрывание по x
    dx = min_clip_x - xstart;

    // Изменение значений интерполируемых
    // величин
    ui+=dx*du;
    vi+=dx*dv;
    wi+=dx*dw;

    // Задаем исходные значения переменных
    xstart = min_clip_x;
} // if

// Проверяем, нужно ли отсекать по x правую
// часть строки
if (xend > max_clip_x)
    xend = max_clip_x;

////////////////////////////////////

// Выводим строку
for (xi=xstart; xi<=xend; xi++)
{
    // При выводе предполагаем, что цвет
    // задан в режиме 5.6.5
    screen_ptr[xi] =
        ((ui >> (FIXP16_SHIFT+3)) << 11) +
        ((vi >> (FIXP16_SHIFT+2)) << 5) +
        (wi >> (FIXP16_SHIFT+3));

    // Интерполируем u,v
    ui+=du;
    vi+=dv;
    wi+=dw;
} // for xi

// Интерполируем u,v,w,x вдоль правой и
// левой сторон

```

```

        xl+=dxdyl;
        ul+=dudyl;
        vl+=dvdyL;
        wl+=dwdyl;
        xr+=dxdyr;
        ur+=dudyr;
        vr+=dvdyr;
        wr+=dwdyr;
        // Сдвиг указателя на точку на экране
        screen_ptr+=mem_pitch;
    } // for y
} // if clip

// Для экономии места ОПУЩЕН ЧАСТНЫЙ СЛУЧАЙ БЕЗ
// ОТСЕЧЕНИЯ

} // if

// Для экономии места ОПУЩЕН ОБЩИЙ СЛУЧАЙ

} // Draw_Gouraud_Triangle16

```

Функция `Draw_Gouraud_Triangle16()`, с которой мы только что ознакомились, выполняет затенение по Гуро в 16-битовом режиме. В ней отсутствуют какие бы то ни было концепции освещения; все, что она делает, — это интерполяция цветов вершин, хранящихся в элементах массива `lit_color[]`, по внутренней области треугольника. А теперь у меня для вас плохие новости: во всех фрагментах программ, в которых важна скорость, с этого момента я решил использовать формат 5.6.5. Другими словами, я не хочу терять скорость, проверяя, в каком режиме мы работаем, или применяя указатели для доступа к функции, работающей с правильной битовой глубиной. Это не должно вызывать никаких проблем, поскольку 99,9% видеокарт работают в режиме 5.6.5. Однако если так случилось, что вам попала карта, работающая в режиме 5.5.5, фрагменты кода, выполняющие манипуляции с битами, можно переписать. Например, рассмотрим внутренний цикл, взятый из приведенной выше функции, в котором производится растеризация по горизонтали.

```

// Выводим строку
for (xi=xstart; xi<=xend; xi++)
{
    // При выводе предполагается,
    // что цвета заданы в формате 5.6.5
    screen_ptr[xi] - ((ui >> (FIXP16_SHIFT+3)) << 11) +
        ((vi >> (FIXP16_SHIFT+2)) << 5) +
        (wi >> (FIXP16_SHIFT+3));
    // Интерполируем значения u,v,w
    ui+=du;
    vi+=dv;
    wi+=dw;
} // for xi

```

Внимательно изучив этот код, можно заметить, что в нем вручную сформирован пиксель в формате 5.6.5. Это делается таким образом, потому что здесь крайне нежелательно вызывать какие бы то ни было функции. Если нужно изменить режим, то соответствующе-

шие изменения необходимо будет внести везде, где производится вывод на экран. Это будет легко, поскольку сопровождается операциями побитового сдвига и наложения маски. Кроме того, такие фрагменты везде снабжены комментариями.

**НА ЗАМЕТКУ**

Возможно, вы удивлены тем, что переменные, в которых хранятся интерполируемые величины, получили имена `u`, `v` и `w`. На самом деле, эта функция первоначально предназначалась для наложения текстуры. Затем, вместо того, чтобы переименовывать эти переменные в `g`, `g` и `b`, я использовал старые переменные `u` и `v`, а также ввел новую переменную `w`. Таким образом, переменные `u`, `v` и `w` соответствуют красному, зеленому и синему цветам. Это просто имена переменных, но если они сбивают вас с толку, измените их сами! Просто эти переменные встречаются так часто, что, даже воспользовавшись средствами автоматического поиска и замены, легко допустить ошибку.

Теперь посмотрим на демонстрационную программу `DEMO19_1.CPP|EXE`, выполняющую затенение по Гуро. Она не делает ничего особенного, просто выводит треугольники. Копия экрана, полученная в результате работы программы `DEMO19_1.CPP|EXE`, приведена на рис. 9.26. Попробуйте запустить программу на своем компьютере. В ходе ее работы случайным образом выбираются положения и цвета вершин треугольника, а затем вызывается функция, выполняющая затенение треугольника по Гуро. Имеется также 8-битовая версия этой функции с именем `DEMO19_1_8b.CPP|EXE`.

## Добавление освещения вершин в функцию, выполняющую затенение по Гуро

Следующий шаг будет состоять в том, чтобы с помощью функций, входящих в состав процессора освещения, вычислить фактические интенсивности освещения в каждой вершине и получить конечные цвета вершин в формате RGB. Для этого достаточно вместо стандартной функции для плоского затенения вызвать функцию, выполняющую затенение по Гуро. В результате в поле зрения будут отражаться треугольники с затенением по Гуро, освещение которых можно будет менять. Перечислим, что нужно для выполнения этой задачи. Новые структуры для треугольников, в которых содержатся массивы, предназначенные для хранения конечных цветов с учетом освещения, уже созданы. Упомянутый массив имеет имя `lit_color[]` и содержит три сегмента, по одному для каждой вершины треугольника. Таким образом, все, что нужно сделать, — взять функцию, в которой моделируется освещение на уровне списка визуализации или на уровне самого объекта, и изменить ее так, чтобы она научилась обрабатывать многоугольники с излучательным затенением, плоским затенением и затенением по Гуро.

У нас уже есть такая функция, способная обрабатывать треугольники с излучательным и плоским затенением. В случае излучательного затенения в этой функции не выполняется никаких действий; просто значение поля с цветом многоугольника непосредственно копируется в элемент массива `lit_color[0]` — и все. Однако для треугольников с плоским затенением функция обрабатывает каждый активный источник освещения и моделирует освещение многоугольника на основе его базового цвета и нормали к поверхности. Таким образом, просто нужно перенести эти функциональные возможности в другой цикл, чтобы многоугольники с затенением по Гуро освещались в вершинах. Этот процесс схематически показан на рис. 9.27, на котором изображены многоугольники, проходящие по конвейеру игрового процессора, процессор освещения, в котором для каждого многоугольника вычисляются параметры освещения, а также указаны этапы, из которых состоит весь процесс.

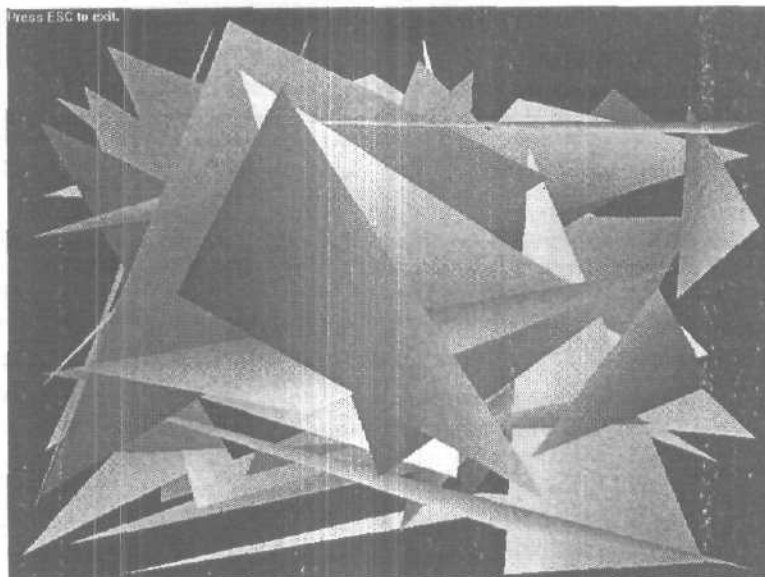


Рис. 9.26. Копия экрана при работе функции, которая визуализирует треугольники с затенением по Гуро



Рис. 9.27. Новый конвейер **освещения**, поддерживающий как плоское затенение, так и затенение по Гуро

Как видно из рисунка, нам нужно воспроизвести код освещения для треугольников с затенением по Гуро. При этом следует принять во внимание, что нормали таких треугольников хранятся в поле  $n$  (нормаль) структуры VERTEX4DTV1, и это почти все! Единственный недостаток — тот, что для каждого треугольника нужно выполнить трехкратные вычисления, поскольку необходимо учесть **освещение** всех вершин! Однако здесь **можно** провести некоторую оптимизацию, которую мы обсудим далее в этом разделе.

Рассмотрим новые функции **освещения**, поддерживающие освещение вершин и затенение по Гуро в 16- и 8-битовых режимах.

## Освещение по Гуро в 16-битовом режиме

Как уже упоминалось, добавить поддержку **освещения** вершин в функциях освещения достаточно легко. Все сводится к тому, чтобы скопировать код и заменить нормали к поверхностям многоугольников, которые используются при плоском затенении, нормальными, проведенными в вершинах, и троекратными вычислениями, связанными с освещением, по одному на каждую вершину. Единственная проблема в том, что из-за этих изменений функции освещения становятся слишком сложными, и выполнение **каждой** функции может занять много времени. Вот почему существует не так уж много программно-ориентированных полноцветных систем освещения с затенением по Гуро. Однако получившаяся система не так уж и плоха, и ее производительность вполне достаточна для игр.

Чтобы реализовать освещение по Гуро, нам понадобится переписать две функции; в одной из них освещение моделируется на уровне объекта, а в другой — на уровне списка визуализации. Я собираюсь продемонстрировать версию, работающую со списком визуализации, потому что именно на уровне списка визуализации проделана большая часть работы, включая освещение. Несмотря на то, что в предыдущей версии игрового процессора реализованы функции освещения, работающие на уровне объекта и на уровне списка визуализации, а также в 8- и 16-битовых режимах, мы предпочитаем использовать список визуализации. Однако освещению по Гуро присущи некоторые особенности, обусловившие необходимость оптимизации процесса.

Дело вот в чем. Взгляните на рис. 9.28. На нем изображен куб в виде цельного объекта и в виде набора многоугольников в списке визуализации. Проблема в том, что когда куб представлен в виде объекта `OBJECT4DV2`, у него есть всего 8 нормалей, проведенных в вершинах (поскольку вершин только 8). Но как только мы разделим куб и поместим его составляющие в список визуализации, то получим  $12 \cdot 3 = 36$  нормалей! Конечно же, причиной этого является тот факт, что каждый многоугольник представляет собой отдельный самосогласованный элемент списка визуализации и все взаимосвязи между многоугольниками опускаются.

Выполняя освещение на уровне объекта, то есть на той стадии, когда он **представляет** собой неразделенный каркас, мы получаем преимущество за счет минимального количества нормалей в вершинах. Если же мы подождем, пока объект будет преобразован в набор многоугольников, а затем выполним его освещение в списке визуализации, то количество вычислений возрастет в несколько раз из-за необходимости обработки нескольких копий одних и тех же **вершин**. Конечно же, это недопустимо, и мы еще **вернемся** к этой теме позже, на стадии оптимизации.

На данном этапе функции освещения, работающие как на уровне объекта, так и на уровне списка визуализации, выполняют освещение трех вершин для **каждого** многоугольника. Даже если бы функция, работающая с объектом, могла каким-то образом отмечать уже обработанные вершины, чтобы свести вычисления к минимуму, в первой версии добиться этого будет весьма сложно. Поэтому я хотел бы отложить решение этого вопроса, но и забывать о нем мы не будем.

Причина, усложняющая однократную обработку всех вершин, заключается в том, что при освещении вершины и сохранении ее конечного цвета в элементах массива `lit_colors[]` для вычислений цвета используется основной цвет многоугольника, в который входят эти вершины. Поэтому для повторного использования уже освещенных вершин процедуру освещения нужно разбить на два этапа. На первом этапе вычисляется **только** интенсивность освещения, после чего **освещенная** вершина помечается и больше не обрабатывается, если она встретится в другом треугольнике, имеющем с текущим **общие** вершины. Далее следует этап последующей обработки, когда предварительно **получен-**

ные интенсивности освещения модулируются фактическими цветами многоугольников. Детали, возникающие в ходе реализации этого плана, усложняют код, а на данный момент нам и так хватает работы. Однако эта оптимизация будет выполнена в последующих главах, поэтому будьте внимательны. При желании вы можете сами это сделать.

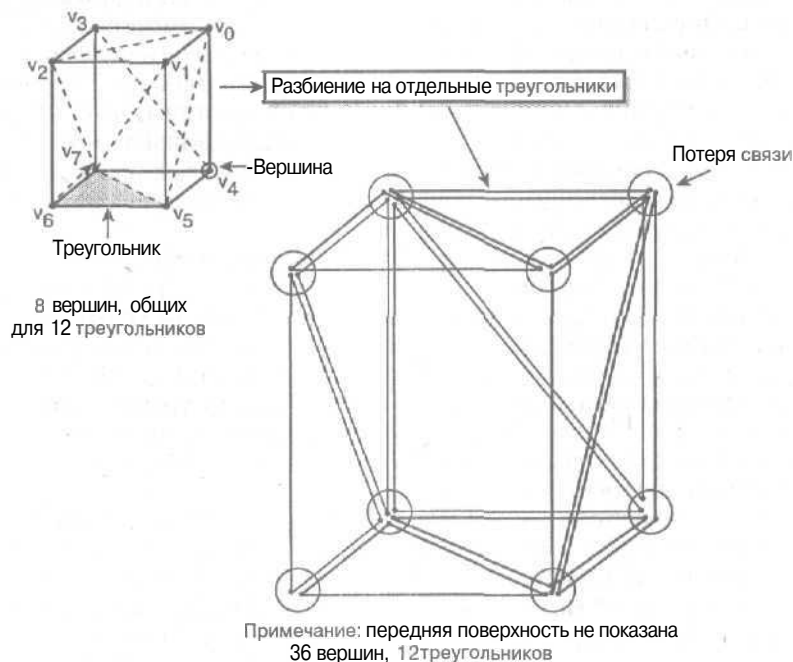


Рис. 9.28. Цельный объект имеет вершины, общие для нескольких граней, однако при разбиении на многоугольники эта связь теряется

Рассмотрим функции, с помощью которых моделируется освещение в 16-битовом режиме. Исходный код этих функций содержится на прилагаемом компакт-диске. Они слишком объемные, чтобы приводить их листинги в этой книге (более 30 страниц), однако новых среди них всего две. Приведем их прототипы.

```
int Light_RENDERLIST4DV2_World16(
    RENDERLIST4DV2_PTR rend_list, // Список
    CAM4DV1_PTR cam,              // Положение камеры
    LIGHTV1_PTR lights,            // Список источников
                                  // освещения
    int max_lights)                // Максимальное количество
                                  // источников в списке.
```

Существует также версия этой функции, работающая с объектом.

```
int Light_OBJECT4DV2_World16(
    OBJECT4DV2_PTR obj,           // Обрабатываемый объект
    CAM4DV1_PTR cam,              // Положение камеры
    LIGHTV1_PTR lights,            // Список источников
                                  // освещения
    int max_lights)               // Максимальное количество
                                  // источников в списке.
```

Эти функции работают точно так же, как и их предыдущие версии, рассмотренные в последней главе. Изменениям подвергся сам список визуализации и формат объекта, и более ничего нового в вызове этих функций нет. Однако на фрагментах кода, извлеченных из них, я хотел бы продемонстрировать некоторые интересные добавления, касающиеся реализации затенения по Гуро. Как я уже говорил, эти функции слишком объемны, чтобы приводить их листинги полностью, однако некоторые их части заслуживают внимания. Ниже приведен пример кода, в котором выполняются вычисления по освещению объекта *общими* и бесконечно удаленными источниками. В той части, где реализуется затенение по Гуро, расчеты проводятся на основе параметров вершин.

```
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
{
    // Затенение по Гуро. К сожалению, на этом этапе
    // информация о связях в исходном каркасе утеряна, и он
    // разбит на треугольники. Поскольку есть много
    // треугольников с общими вершинами, то такие вершины
    // будут освещаться по несколько раз. К сожалению, нет
    // возможности это предотвратить. Приходится признать,
    // что освещение с затенением по Гуро лучше выполнять на
    // уровне объекта, поскольку сохраняется информация о
    // совместных вершинах. Здесь реализация освещения
    // аналогична случаю плоского затенения, однако она
    // осуществляется три раза, по количеству вершин. Есть
    // много возможностей для оптимизации, однако на данном
    // этапе мы их отложим, чтобы получить понятный код.
    // Позже мы вернемся к этим возможностям

    // Этап 1: извлечение базового цвета в режиме RGB;
    // предполагается, что он задан в формате 565
    _RGB565FROM16BIT(curr_poly->color, &r_base,
                     &g_base, &b_base);

    // Преобразование к 8-битовому формату
    r_base <<= 3; g_base <<= 2; b_base <<= 3;

    // Инициализация цветов вершин
    r_sum0 = 0; g_sum0 = 0; b_sum0 = 0;
    r_sum1 = 0; g_sum1 = 0; b_sum1 = 0;
    r_sum2 = 0; g_sum2 = 0; b_sum2 = 0;

    // Новая оптимизация:
    // Если в системе несколько источников, придется
    // выполнять многочисленные ненужные вычисления. Чтобы
    // свести их к минимуму, я применил такую стратегию. В
    // процессе вычислений, связанных с моделированием
    // освещения, переменным ключа в каждом цикле
    // присваиваются значения MAX. Далее проверяем, равны ли
    // эти переменные значению MAX. Если это так, то
    // обрабатывается первый источник, для которого
    // требуются математические вычисления. Затем эта
    // информация сохраняется в переменной (при этом ее
    // недопустимое значение изменяется). Далее
    // обрабатываются другие источники, для которых
    // требуются математические вычисления, с использованием
    // предварительно полученного значения. С этой целью
```

```

// организуется цикл по источникам
for (int curr_light - 0; curr_light < max_lights;
    curr_light++)
{
    // Если источник активен
    if (lights[curr_light].state == LIGHTV1_STATE_OFF)
        continue;

    // Проверка типа источника
    if (lights[curr_light].attr & LIGHTV1_ATTR_AMBIENT)
    {
        // Умножаем значения каждого канала на цвет
        // многоугольника, а затем делим результат на
        // 256, чтобы преобразовать его обратно к
        // интервалу 0..255. В реальности следует
        // использовать сдвиг >> 8
        ri = ((lights[curr_light].c_ambient.r * r_base)
            / 256);
        gi = ((lights[curr_light].c_ambient.g * g_base)
            / 256);
        bi = ((lights[curr_light].c_ambient.b * b_base)
            / 256);

        // Источник общего освещения воздействует на
        // каждую вершину таким же образом
        r_sum0 += ri; g_sum0 += gi; b_sum0 += bi;
        r_sum1 += ri; g_sum1 += gi; b_sum1 += bi;
        r_sum2 += ri; g_sum2 += gi; b_sum2 += bi;
    } // if
    else if (lights[curr_light].attr &
        LIGHTV1_ATTR_INFINITE)
    {
        // Для бесконечно удаленного источника
        // понадобится нормаль к поверхности и
        // направление на источник. Нет необходимости
        // вычислять параметры нормали в вершине,
        // поскольку они уже известны, а длина нормали
        // нормирована на 1
        // ...
        // Напомним вид математической модели бесконечно
        // удаленного источника освещения:
        //  $I(d)dir = I_0dir * Cldir$ 
        // и модели диффузионного источника:
        //  $I_{totald} = R_{sdiffuse} * I_{diffuse} * (\pi L)$ 
        // Итак, нужно перемножить все эти величины.
        // Обратите внимание на масштабный множитель
        // 128. Хотелось бы избежать операций над
        // числами с плавающей точкой - не потому, что
        // они выполняются медленнее, а потому, что
        // преобразования типов занимают время

        // Освещение нужно выполнить для всех вершин

        // Вершина 0
        dp = VECTOR4D_Dot(&curr_poly->tvlist[0].n,

```

```

        &lights[curr_light].dir);

// Источник добавляется только при условии dp>0
if (dp > 0)
{
    i = 128*dp;
    r_sum0 += (lights[curr_light].c_diffuse.r *
               r_base * i) / (256*128);
    g_sum0 += (lights[curr_light].c_diffuse.g *
               g_base * i) / (256*128);
    b_sum0 += (lights[curr_light].c_diffuse.b *
               b_base * i) / (256*128);
} // if

// Вершина 1
dp = VECTOR4D_Dot(&curr_poly->tvlist[1].n,
                  &lights[curr_light].dir);

// Источник добавляется только при условии dp>0
if (dp > 0)
{
    i = 128*dp;
    r_sum1 += (lights[curr_light].c_diffuse.r *
               r_base * i) / (256*128);
    g_sum1 += (lights[curr_light].c_diffuse.g *
               g_base * i) / (256*128);
    b_sum1 += (lights[curr_light].c_diffuse.b *
               b_base * i) / (256*128);
} // if

// Вершина 2
dp = VECTOR4D_Dot(&curr_poly->tvlist[2].n,
                  &lights[curr_light].dir);

// Источник добавляется только при условии dp>0
if (dp > 0)
{
    i = 128*dp;
    r_sum2 += (lights[curr_light].c_diffuse.r *
               r_base * i) / (256*128);
    g_sum2 += (lights[curr_light].c_diffuse.g *
               g_base * i) / (256*128);
    b_sum2 += (lights[curr_light].c_diffuse.b *
               b_base * i) / (256*128);
} // if

} // if (источник бесконечно удаленный)

```

Ознакомившись с этим кодом, можно заметить, что для хранения интенсивностей красного, зеленого и синего каналов, соответствующих вершинам 0..2, в нем используются переменные `r_sum0..2`, `g_sum0..2`, `b_sum0..2`. Базовый цвет освещаемого треугольника находится в переменных `r_base`, `g_base` и `b_base`. Вычисления имеют такой же характер, как и при плоском затенении, но они производятся три раза. Чтобы подчеркнуть это различие, в той части, в которой моделируется бесконечно удаленный источник, я выделил строки, где начинаются вычисления для каждой вершины.

В этой версии функции **освещения** применяется очень интересная оптимизация, которую я назвал *кэшированием вычислений* (calculation caching). По сути, она основана на анализе циклов освещения. В процессе освещения каждого многоугольника происходит обработка его поверхности для различных источников освещения: общих, бесконечно удаленных, точечных, световых пятен и т.д. При этом все величины вычисляются заново. Например, если нам понадобилось **расстояние** от источника освещения до многоугольника, мы вычисляем его; понадобилась нормаль к поверхности, мы вычисляем и ее и т.д. Трюк состоит в том, чтобы определить, какие величины **инвариантны** относительно источника освещения, и сохранить их в памяти. Скажем, параметры нормали к многоугольнику не зависят от того, каким источником освещается этот многоугольник. Следовательно, эту величину можно рассматривать в качестве кандидата для кэширования. Это замечательный метод, позволяющий ускорить процесс **освещения**, однако он работает только при наличии нескольких источников. Я не слишком углублялся в этот вопрос и реализовал кэширование только для нормалей к поверхностям, которые имеют общий вид при моделировании точечных источников и световых пятен. Позже мы еще вернемся к этой оптимизации и поработаем над ней более тщательно.

К сожалению, при затенении по Гуро подобным трюком воспользоваться не удастся; это возможно только при плоском затенении. Ниже приведен фрагмент оптимизированного кода из той части, в которой реализуется плоское затенение. Его начало находится непосредственно перед циклом, в котором обрабатывается каждый источник **освещения**.

```
// Поверхности normal.z присваивается значение FLT_MAX,
// чтобы пометить ее как необработанную
n.z = FLT_MAX;
```

```
// Цикл по источникам освещения
for (int curr_light=0; curr_light<max_lights; curr_light++)
{
```

Одной из компонент вектора нормали присваивается значение, соответствующее бесконечности, чтобы пометить ее как некорректную. Затем начинается цикл по источникам освещения. Модели освещения, для которых требуется информация о нормали, содержатся в следующем коде.

```
// Проверяем, вычислена ли ранее нормаль к многоугольнику
if (n.z==FLT_MAX)
{
    // Для этого многоугольника нужно вычислить параметры
    // нормали к его поверхности. Напомним, что вершины
    // упорядочены по ходу часовой стрелки,
    // u=p0->p1, v=p0->p2, n=uxv

    // Находим координаты векторов u и v
    VECTOR4D_Build(&curr_poly->tvlist[0].v,
        &curr_poly->tvlist[1].v, &u);
    VECTOR4D_Build(&curr_poly->tvlist[0].v,
        &curr_poly->tvlist[2].v, &v);

    // Вычисляем векторное произведение
    VECTOR4D_Cross(&u, &v, &n);
} // if
```

Сначала тестируется значение переменной n.z. Если оно все еще равно FLT\_MAX, то вычисляются координаты вектора нормали; в противном случае вычисления опускаются.

Таким образом, даже для восьми источников освещения нормаль к каждому многоугольнику вычисляется только один раз. Правда, здорово?

**СОВЕТ**

Такая стратегия может привести к значительному повышению производительности программы. Однако в своих ожиданиях можно зайти слишком далеко, предположив, что в игре есть множество источников освещения, а их на самом деле всего один или два. В этом случае описанная оптимизация только замедлит работу программы. Таким образом, применяя такую оптимизацию, убедитесь, что необходимые для нее дополнительные затраты ресурсов окупятся благодаря наличию достаточного количества источников освещения.

Ознакомьтесь с демонстрационной программой DEMOII9\_2.CPP[EXE], в которой процессор освещения генерирует параметры источников освещения и производится затенение по Гуро. На рис. 9.29 приведен снимок экрана, полученный в результате ее работы. В этой программе создаются три модельные изображения молекул воды, которые затем вращаются, освещаемые несколькими источниками. Одна из этих молекул имеет постоянное затенение, другая — плоское затенение и последняя — затенение по Гуро. Попробуйте угадать, какому из изображений какое затенение соответствует. Обратите внимание, каким округлым выглядит изображение молекулы с затенением по Гуро, несмотря на то, что это все граненые модели. Конечно же, это результат усреднения вершин на стыке треугольников.

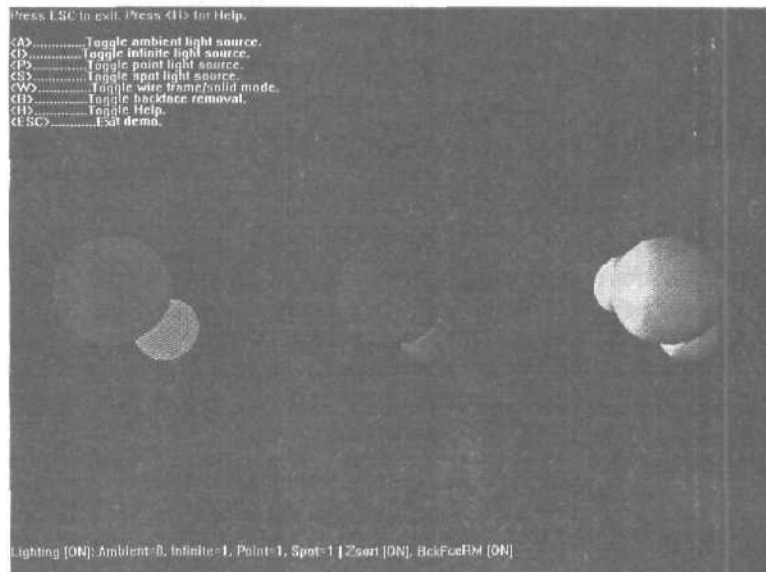


Рис. 9.29. Полнофункциональная трехмерная демонстрационная программа с освещением по Гуро

## Освещение по Гуро в 8-битовом режиме

В этом случае начинаются небольшие неприятности. Неожиданно оказывается, что освещение в 8-битовом режиме работает почти так же, как и в 16-битовом. Фактически, в обеих версиях функций освещения используется почти один и тот же код. Единственное различие заключается в том, что если многоугольник имеет излучательное или плоское затенение, индекс конечного цвета вычисляется и заносится в элемент массива `lit_color[0]`, а обращение к таблице соответствия выглядит следующим образом.

```
curr_poly->lit_color[0] -
    rgblookup[RGB16Bit565(r_sum, g_sum, b_sum)];
```

Напомним, как работает **таблица** соответствия. В качестве ее 16-битовых индексов используются RGB-слова в формате 5.6.5 (полученные путем соединения). На выходе таблицы получаем индекс цвета, который лучше всего соответствует входному цвету в формате RGB. Если вы освежите в памяти материал предыдущей главы, то **вспомните**, что таблица соответствия создается на этапе инициализации с **помощью** следующего вызова.

```
// Создание таблицы соответствия для процессора освещения
RGB_16_8_IndexedRGB_Table_Builder(
    DD_PIXEL_FORMAT565, // Формат пикселя
    palette,             // Исходная палитра
    rgblookup);          // Хранилище таблицы.
```

Казалось бы, все хорошо, но проблема в том, что конечные цвета вершин, входящих в состав треугольника с затенением по Гуро, нет смысла преобразовывать в индексы цветов. Ведь тогда мы не сможем выполнять интерполяцию! Интерполировать можно только цвета, выраженные в формате RGB. Поэтому при **освещении** треугольника с затенением по Гуро в 8-битовом режиме в каждый из элементов массива `lit_colors[0..2]` заносятся значения цветов в формате RGB, то есть все происходит так же, как и в 16-битовой версии. На выходе таблицы получаем индекс цвета, который лучше других соответствует запрашиваемому цвету в формате RGB. Прототип функции, выводящей треугольники с затенением по Гуро в 8-битовом режиме, выглядит следующим образом.

```
void Draw_Gouraud_Triangle(
    POLYF4DV2_PTRface, // Указатель на поверхность
    UCHAR*dest_buffer, // Указатель на видеобuffer
    int mem_pitch)      // Количество байтов в строке,
                        // 320, 640 и т.д.
```

Все выглядит так же, как и в 16-битовой версии. Отличия проявляются только в задании начальных значений интерполируемых величин и выводе конечных пикселей на экран. Функция слишком объемна, поэтому мы не станем приводить ее листинг, однако ознакомимся с некоторыми ее фрагментами. Во-первых, приведем код, с помощью которого извлекается начальный цвет в формате RGB.

```
// Предполагается, что цвет задан в формате 5.6.5. К
// сожалению, мы не можем позволить себе вызывать функции во
// внутреннем цикле, поэтому для поддержки форматов 5.6.5 и
// 5.5.5 необходимо создать две версии с жестко
// закодированными форматами. Обратите внимание: несмотря на
// то, что растеризация выполняется в 8-битовом режиме,
// входные данные представляются в формате RGB
_RGB565FROM16BIT(face->lit_color[v0], &r_base0, &g_base0,
    &b_base0);
_RGB565FROM16BIT(face->lit_color[v1], &r_base1, &g_base1,
    &b_base1);
_RGB565FROM16BIT(face->lit_color[v2], &r_base2, &g_base2,
    &b_base2);

// Преобразование в 8-битовый режим
r_base0 <=> 3;
g_base0 <=> 2;
b_base0 <=> 3;
```

```
// Преобразование в 8-битовый режим
r_base1 <=< 3;
g_base1 <=< 2;
b_base1 <=< 3;
```

```
// Преобразование в 8-битовый режим
r_base2 <=< 3;
g_base2 <=< 2;
b_base2 <=< 3;
```

Внимательно изучив этот фрагмент кода, можно заметить, что он такой же, как и в 16-битовой версии! Так и должно быть, поскольку в 8-битовых версиях функций освещения, обрабатывающих многоугольники с затенением по Гуро, конечные цвета вершин сохраняются в элементах массива `lit_colors[0..2]` не в виде индексов цвета, а в виде слов в формате RGB 5.6.5. Преобразование в индексы цвета привело бы к бесполезной трате времени и ресурсов, поскольку индексы невозможно интерполировать. Самое важное происходит в завершающем внутреннем цикле, в котором осуществляется растеризация строк развертки. Рассмотрим соответствующий фрагмент кода.

```
// Вывод строки развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Предполагаем, что цвета пикселей заданы в формате
    // 5.6.5
    screen_ptr[xi] =
        rgblookup[(((ui >> (FIXP16_SHIFT+3)) << 11) +
            ((vi >> (FIXP16_SHIFT+2)) << 5) +
            (wi >> (FIXP16_SHIFT+3)))];
    // Интерполяция значений u и v
    ui+=du;
    vi+=dv;
    wi+=dw;
} // for xi
```

Что же происходит в этом фрагменте? В нем RGB-значения, разбросанные по трем каналам, преобразуются в единый входной RGB-индекс таблицы соответствия, с помощью которого производится поиск индекса цвета, наиболее соответствующего данному цвету.

В результате всех этих обсуждений возникает вопрос: не получится ли, что 8-битовая версия работает медленнее, чем 16-битовая? На этот вопрос нет однозначного ответа. Ответ будет положительным, если производится полноцветное освещение; если же используется монохроматическое освещение, которое особенно хорошо подходит для 8-битового режима, то ответ будет отрицательным. Все довольно просто. Если программа работает на очень медленном компьютере, то о выводе 16-битовой графики не может быть и речи. То же самое можно сказать о цветном освещении (или о его разновидностях). В таком случае придется ограничиться только поддержкой источников белого цвета, и функция затенения по Гуро будет интерполировать только интенсивности. Затем можно создать таблицу соответствия, содержащую 256 цветов и 256 оттенков каждого цвета, после чего — проиндексировать элементы таблицы значениями цветов и интенсивностей. Так получилось, что у нас уже есть такая таблица именно для этого случая. При желании можно было бы написать 8-битовые версии функций, с помощью которых моделируется освещение, и сделать так, чтобы их выходным значением была бы только интенсивность. Тогда в них выполнялась бы интерполяция только по одному каналу, т.е. по интенсивности, а это работает в три раза быстрее, чем интерполяция по каналам

цвета в формате RGB. Во время **растеризации** можно было бы воспользоваться таблицей соответствия цветов и **интенсивностей**, чтобы найти индекс цвета для каждой комбинации "цвет-интенсивность". Таким образом, мы получили бы быструю версию функции, выполняющую монохроматическое затенение по Гуро в 8-битовом режиме.

Можно было бы подойти творчески и создать для цветных источников **освещения** несколько таблиц соответствия (рис. 9.30). Затем можно было бы пропустить значение интенсивности через одну из цветовых таблиц, сначала выбрав цвет источника, который лучше других **соответствует** цвету освещения, **падающего** на сцену. Идея понятна? Например, предположим, что нам заранее известно, что большинство источников будут белого, красного, зеленого, синего и оранжевого цветов. В этом случае целесообразно создать пять таблиц соответствия, в которых индексы цветов вместе с индексами интенсивности отображаются на индексы тех цветов палитры, тон или цвет которых являются такими, как если бы на поверхность падал свет от источника данного цвета. Этот способ позволяет с помощью таблиц соответствия моделировать освещение цветными источниками, не влияя на производительность.

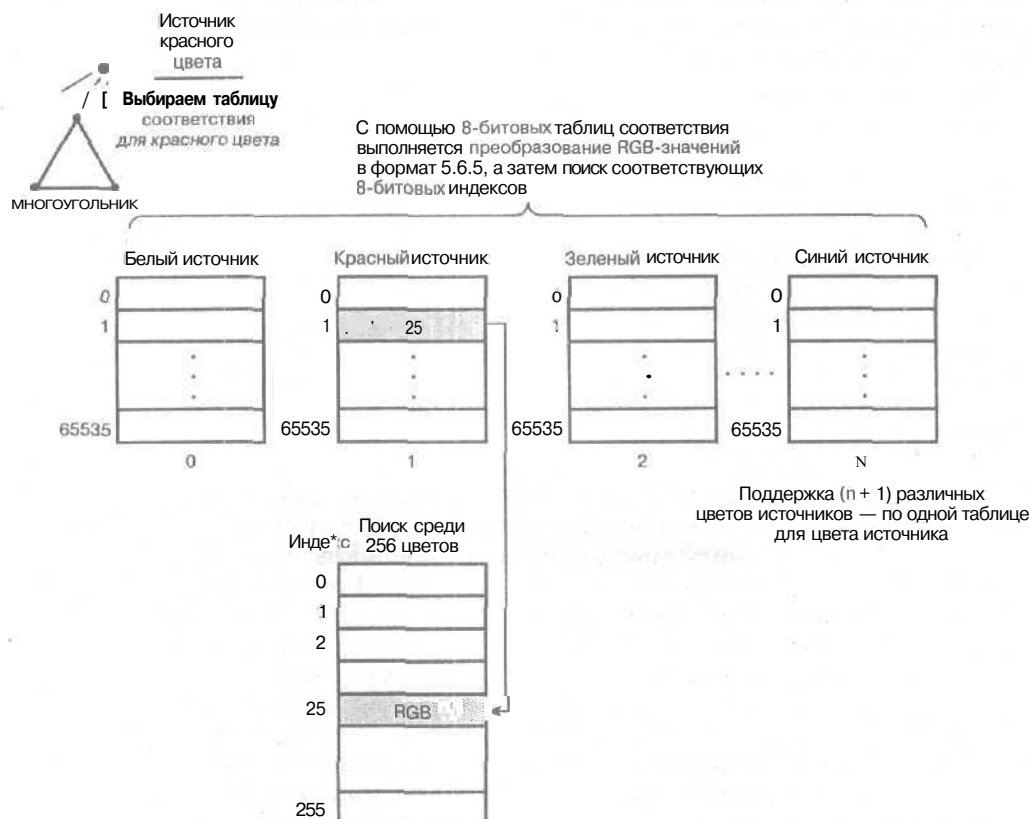


Рис. 9.30. Использование нескольких таблиц соответствия, содержащих **предвычисленные значения интенсивностей** для общих цветов источников

Основной вывод состоит в том, что вся эта работа по реализации 8-битового режима не так уж необходима, поскольку производительности современных компьютеров вполне хватает для функционирования в 16-битовом режиме. Фактически, 8-битовые фрагменты кода

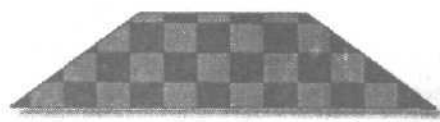
замедляют работу программы, поскольку мы на самом деле нацелены на 16-битовый режим, а затем переходим к 8-битовому режиму! С другой стороны, трюки, подобные использованию таблиц соответствия, все равно могут пригодиться. Ведь никто не сказал, что это же нельзя сделать в 16-битовом режиме. Например, если известно, что мы не собираемся использовать никаких других источников **освещения**, кроме красных, зеленых, синих, белых, оранжевых, ..., можно воспользоваться единой монохроматической интерпретацией в 16-битовой версии программы затенения по Гуро. Для источника, освещающего многоугольник, в качестве входного параметра функции можно выбрать правильную таблицу соответствия и избавиться от двух других интерполируемых величин. Здорово, правда? Мы займемся этим позже, на стадии оптимизации.

Ознакомьтесь с демонстрационной программой `DEMO119_2_8b.CPP|EXE`, работающей в 8-битовом режиме. Запустите ее и посмотрите, что происходит. В этой программе создаются три модельные изображения молекул воды, которые затем вращаются, **освещаемые** несколькими источниками. Одна из этих молекул имеет постоянное **затенение**, другая — плоское затенение и **последняя** — затенение по Гуро. Обратите внимание, каким округлым выглядит изображение молекулы с затенением по Гуро, несмотря на то, что все модели одинаковые. Поразительно, что, несмотря на ограниченность цветового пространства, изображение выглядит не так уж плохо.

Но что-то мы задержались на этой теме, а нам пора уже переходить к отображению текстур.

## Основы теории дискретизации

Вы не поверите, но для отображения текстур у нас уже все готово. В этом нет ничего особенного. Просто вместо интерполяции цвета интерполируются координаты текстуры. Конечно же, отображению текстуры присущи свои особенности, но в основном оно очень похоже на затенение по Гуро. На данном этапе речь идет только об *аффинном* или *линейном* отображении текстуры; поправки с учетом перспективы — это совсем другая история. На рис. 9.31 приведен классический пример с шахматной доской, на которую отображается текстура с помощью линейного отображения и с поправкой на перспективу.



а) Аффинное отображение текстуры; обратите внимание на отсутствие перспективы



б) Отображение текстуры с учетом перспективы; обратите внимание на различия между ближними и дальними элементами

*Рис. 9.31. Линейное отображение текстуры и отображение с учетом перспективы*

## Дискретизация в одном измерении

Мы стремимся разработать алгоритм, позволяющий выполнять дискретизацию текстурной карты, чтобы полученные в результате пиксели можно было использовать для определения цветов каждой строки развертки, формируемой в процессе визуализации текущего треугольника. На рис. 9.32 проиллюстрирован этот процесс для треугольников трех основных видов: треугольника с горизонтальной верхней стороной, горизонтальной нижней стороной и треугольника общего вида.



Рис. 9.32. Отображение текстуры для треугольников трех основных видов

На рис. 9.33 подробно продемонстрирован процесс, который мы хотим реализовать. Нужно отобразить текстурную карту прямоугольной формы размером  $m \times m$  пикселей (с цветом, выраженным в 8- или 16-битовом режиме) на произвольный треугольник с определенными координатами. Это означает, что нам нужен способ, позволяющий вращать и изменять масштаб рассматриваемого треугольника. Чтобы облегчить разработку алгоритма, на рис. 9.33 обозначены некоторые интересные нас точки.



Рис. 9.33. Подробная схема отображения текстуры на треугольник в пространстве координат  $u$  и  $v$

Во-первых, заметим, что в состав треугольника, на который накладывается текстура, входят три вершины  $P_0$ ,  $P_1$  и  $P_2$  с координатами  $(x_0, y_0)$ ,  $(x_1, y_1)$  и  $(x_2, y_2)$ , соответственно. Кроме того, оси координатного пространства, в котором построена карта текстуры, обозначены через  $u$  и  $v$ , где  $u$  — горизонтальная ось, а  $v$  — вертикальная ось. Обратите внимание, что координаты как по оси  $u$ , так и по оси  $v$  изменяются в интервале от 0 до 63. В данном примере

точка в левом верхнем углу имеет координаты  $(0,0)$ , а в правом нижнем —  $(63,63)$ . Однако во многих случаях эти координаты могут изменяться от 0 до 1 (нормированные координаты) или от 0 до числа, выраженного любой степенью двойки, минус 1.

Линейное отображение текстуры прямоугольной формы на треугольник приводит к интенсивному интерполированию, из-за чего здесь очень легко допустить ошибку и (или) получить медленный алгоритм. Хотя процесс интерполяции уже описывался и для растеризации треугольников, и для затенения по Гуро, мы *никогда* раньше не рассматривали его для того случая, когда нужно дискретизировать данные. Итак, рассмотрим интерполяцию именно с этой точки зрения, то есть с целью дискретизации данных, а не для интерполирования растеризуемых координат или интенсивностей цветов. Начнем с самого простого случая — одномерного.

На рис. 9.34 продемонстрирована работа самого простого в мире устройства, позволяющего отобразить текстуру на вертикальную линию. На этом рисунке изображена карта текстуры, толщина которой равна одному пикселю, а высота — восьми пикселям. Нужно отобразить ее на многоугольник произвольной высоты с *толщиной*, равной одному пикселю. Как это сделать? В этом нам поможет дискретизация.

Нужно разбить на элементы карту текстуры, являющуюся в данном случае битовым образом размером  $1 \times 8$  пикселей, и отобразить ее на многоугольник размером  $1 \times n$  пикселей, где  $n$  может меняться от 1 до бесконечности. Рассмотрим примеры, проиллюстрированные на рис. 9.34.

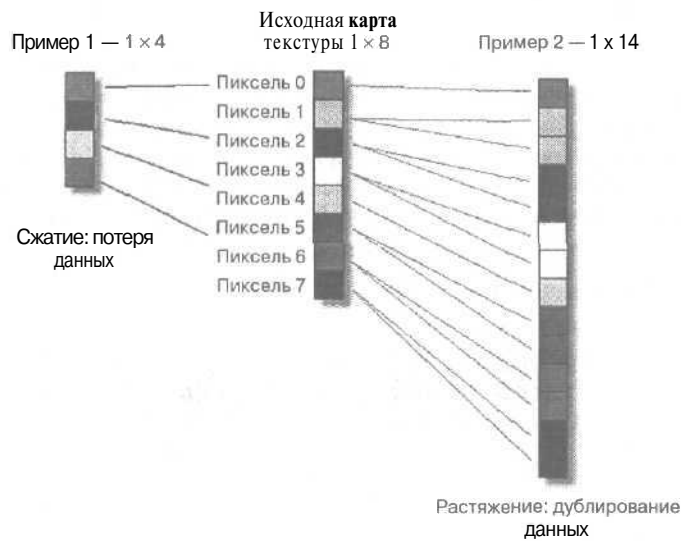


Рис. 9.34. Одномерное отображение текстуры

В качестве первого примера рассмотрим отображение на многоугольник размером  $1 \times 4$  пикселей. Поскольку в нем меньше пикселей, чем в исходной текстуре, то для отображения логично было бы выбрать каждый второй пиксель, как показано на рисунке. Допустим, мы выбрали пиксели  $(0,2,4,6)$  и отображаем их на пиксели многоугольника  $(0,1,2,3)$ . Все замечательно, но как я пришел к выводу, что нужно выбрать именно эти пиксели текстуры? Ответ такой: следует воспользоваться *выборочным отношением* (sampling ratio), которое представляет собой ничто иное, как коэффициент интерполяции.

В общем случае выражение для этой величины имеет **следующий** вид.

$\text{выборочное\_отношение} = \text{исходная\_высота} / \text{целевая\_высота}$

Таким образом, выборочное отношение в рассматриваемом примере представляется **следующим** образом.

$\text{выборочное\_отношение} = 8/4 = 2.0$

Поэтому сдвиг по вертикали в многоугольнике назначения на один пиксель соответствует сдвигу на два пикселя в исходной текстуре. Вот откуда появляется коэффициент 2 и последовательность **(0,2,4,6)**. Если вы следите за ходом изложения, то должны заметить, что при этом теряется **информация** о цветах. Это действительно так, поскольку пришлось отбросить половину пикселей. Если дискретизация целочисленной матрицы выполняется без усреднения и фильтрации, то такая проблема возникает всегда. Создавая высококлассную программу для разработки трехмерных моделей, такую как 3D Studio Max, пришлось бы усреднять **дискретизируемые** пиксели, чтобы добиться лучшего приближения. Однако для игр и программ, работающих в реальном времени, вполне подойдет описанный выше метод (получивший название метода *точечных выборок* (point sampling)). Рассмотрим еще один пример, представляющий другой крайний случай.

В первом примере карта текстуры подвергалась сжатию, то есть многоугольник назначения был меньше исходной текстуры. Из-за этого терялась **информация**. Во втором случае, как нетрудно догадаться, все наоборот, т.е. многоугольник назначения больше, чем исходная текстура, и информации недостаточно. В такой ситуации исходные данные должны будут входить в выборку неоднократно и возникнет необходимость их дублирования. Вот почему в трехмерных играх объекты, которые слишком сильно приближаются, выглядят так неаккуратно. В текстуре содержится недостаточно данных, поэтому элементы, на которые мы ее разбиваем, приходится повторять много раз, и они создают большие блоки. Обратимся ко второму примеру, представленному на рис. 9.34. Как и в **предыдущем** случае, размеры исходной текстуры — 1x8 пикселей, однако целевой многоугольник имеет размеры 1x14 пикселей. Очевидно, выборочное отношение будет выражаться дробным числом. Выполним необходимые математические расчеты.

$\text{выборочное\_отношение} = \text{исходная\_высота} / \text{целевая\_высота}$

Подставляя конкретные числа, получим:

$\text{выборочное\_отношение} = 8/14 = 0.57$

Таким образом, на каждый пиксель назначения приходится по 0.57 выборочных элементов исходной текстуры. В результате получается **представленная** ниже последовательность точечных выборок для целевых пикселей **(0,1,2,...,13)**.

Образец 0:	0.57
Образец 1:	1.14
Образец 2:	1.71
Образец 3:	2.28
Образец 4:	2.85
Образец 5:	3.42
Образец 6:	3.99
Образец 7:	4.56
Образец 8:	5.13
Образец 9:	5.7
Образец 10:	6.27
Образец 11:	6.84
Образец 12:	7.41
Образец 13:	7.98

Чтобы получить номера пикселей, входящих в состав этой последовательности, мы просто отбрасываем дробную часть чисел во втором столбце, или вычисляем значения функции floorQ. В результате получаем последовательность (0,1,1,2,2,3,3,4,5,5,6,6,7,7), которая выглядит вполне правдоподобно. Почти каждый пиксель появляется в ней по два раза, поскольку  $1/0.57 \approx 2$ .

## Билинейная интерполяция

Сделаем небольшое отступление и поговорим об основах фильтрации, а также о том, почему она важна. Как видно из рассмотренного примера, при определении номера пикселя, который должен войти в выборку, просто отбрасывается дробная часть числа с плавающей точкой и оно преобразуется в целое значение. А что было бы, если бы мы принимали во внимание дробную часть и формировали выборочную последовательность из нескольких пикселей, основываясь на геометрическом фильтре? Рассмотрим рис. 9.35. На нем изображен результат наложения текстуры путем точечной выборки, а также ее наложения с помощью фильтрации. Например, двенадцатому образцу из той выборки, которая формировалась в предыдущем примере, соответствует значение 7.41. Однако, если выборка точечная, то в этом значении отбрасывается дробная часть, и в результате получается пиксель текстуры с номером 7. При этом часть информации теряется.

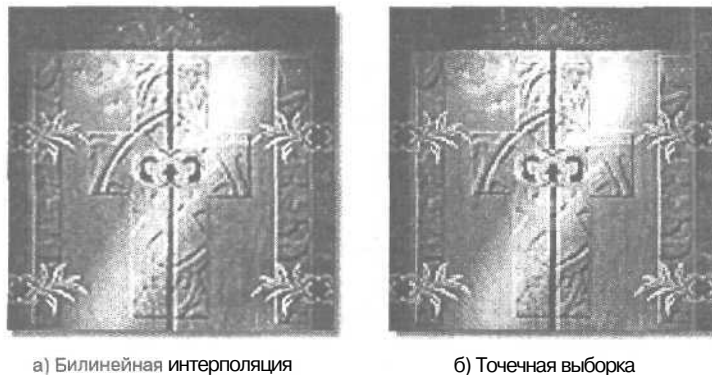


Рис. 9.35. Сравнение результатов точечной выборки и билинейной фильтрации

Конечно же, этот способ работает быстрее, однако изображение, полученное при его применении, выглядит не лучшим образом. Если мы воспользуемся геометрическим фильтром, изображенным на рис. 9.36, то увидим, что пиксель, которому соответствует число 7.41, перекрывается с седьмым и восьмым пикселями. При этом с седьмым пикселем он перекрывается на величину 0.59 (или на 59%), а с восьмым — на величину 0.43 (или на 41%). Напрашивается применение функции смешивания, математическое выражение которой имеет такой вид:

$$\text{результатирующий пиксель текстуры} = (0.59) * \text{пиксель}(7)_{\text{rgb}} + (0.41) * \text{пиксель}(8)_{\text{rgb}}$$

Понятно, что стыковка выполняется в пространстве RGB. Цвет результирующего пикселя представляет собой определенную смесь цветов седьмого и восьмого пикселей. Качество полученного изображения будет намного выше, и в нем исчезнут ступенчатые переходы. Единственный недостаток заключается в том, что этот способ подразумевает дополнительную интерполяцию или, проще говоря, дополнительную операцию стыковки пикселей

вместо обычной выборки. Если это и не означает невозможность реализации в коде программы, то все равно приведет к существенному снижению ее производительности.

Более того, описанный алгоритм нужно реализовать не в одном, а в двух измерениях! Реальная программа для отображения текстуры должна работать в двумерном пространстве, отсюда и термин *билинейная интерполяция* (bilinear interpolation). Как и в других областях, здесь возможны трюки, позволяющие ускорить работу программы с помощью таблиц соответствия или других приближений. Например, если интерполируемые величины выражены в виде чисел с фиксированной точкой, можно отвести, скажем, 2-3 бита для дробной части, которой представляется коэффициент смешивания, и создать таблицу, содержащую 0, 1, 2, ..., 100% каждого цвета. Затем из этой таблицы извлекаются все компоненты смешивания, а результат получается путем их суммирования.

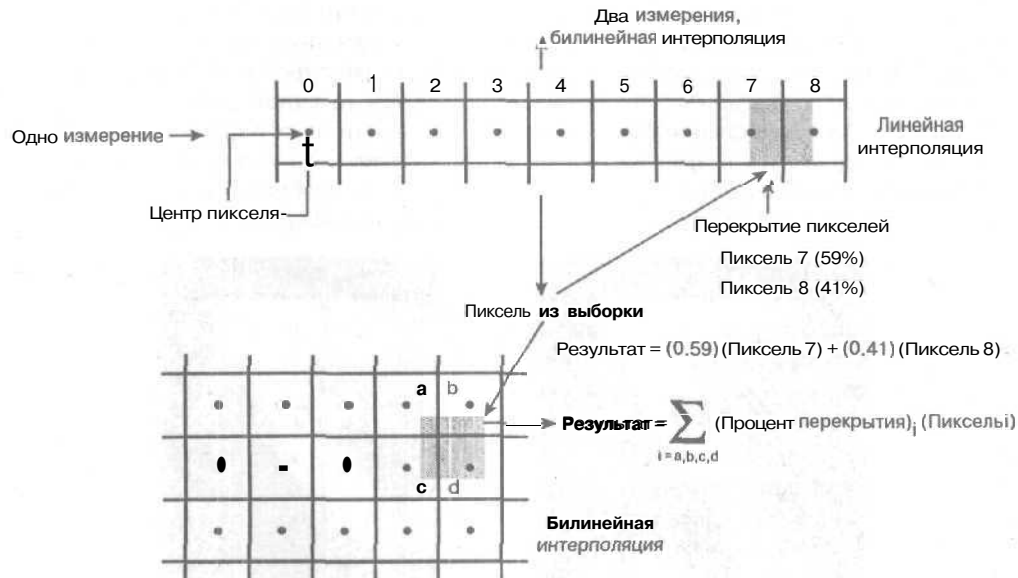


Рис. 9.36. Простой геометрический фильтр для билинейной интерполяции

#### СОВЕТ

Пока что мы не станем реализовывать билинейные фильтры. Возможно это будет сделано позже, когда программа станет достаточно производительной. По сути, фильтрация сводится к сглаживанию текстуры и избавлению от резких переходов, которые получаются при дискретизации. Однако эти эффекты можно свести к минимуму путем предварительного сглаживания используемой в игровом процессоре текстуры. Часть подробностей все равно будет утеряна, однако этот способ заслуживает внимания, и результаты его применения выглядят намного лучше.

## Интерполяция координат и *u* и *v*

Идея, лежащая в основе описываемого алгоритма, состоит в том, чтобы выполнить интерполяцию вдоль левой и правой сторон треугольника, а затем вывести строку развертки, сформированную из *надлежащих* пикселей текстуры. Итак, первое, что нужно сделать, — присвоить вершинам целевого треугольника координаты текстуры и сформировать таким образом каркас для интерполируемых величин. В результате каждая вершина будет характеризоваться четырьмя компонентами (рис. 9.37; координата *z* в расчет не принимается).

Нам понадобится выполнить отображение текстуры на основе величин  $(x, y, u, v)$ . Конечно же, в новом формате, в котором задана структура вершины **VERTEX4DTV1**, уже предусмотрена поддержка координат текстуры, и в состав структуры **OBJECT4DV4** входят массивы  $u$  и  $v$ , предназначенные для этих координат. Итак, у нас есть все необходимое для того, чтобы определить конечный вид текстурного отображения на треугольник.

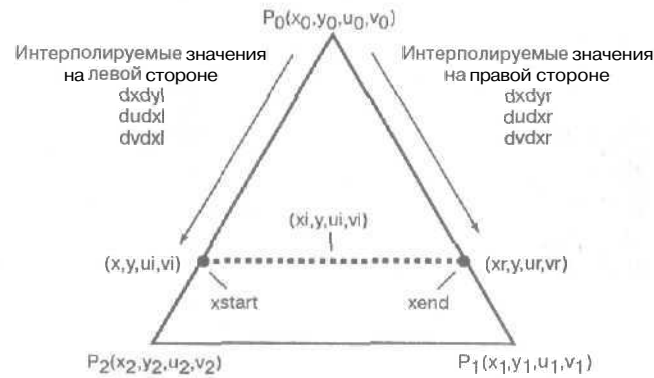


Рис. 9.37. Схема текстурного отображения

Поговорим о том, в каких пределах изменяются координаты текстуры. Поскольку исходная карта текстуры должна иметь размеры  $m \times m$ , где  $m$  — это степень двойки, и чаще всего принимает значения 64, 128 и 256, то координаты текстуры должны изменяться от 0 до  $m-1$ . Для конкретности обсуждения предположим, что все текстуры имеют размеры  $64 \times 64$ . Таким образом, координаты текстуры, соответствующие каждой вершине, должны находиться в пределах  $(0..63, 0..63)$ . При этом карта текстуры отображается (или "натягивается") на каждую вершину.

На рис. 9.38 приведено два примера. На одном из них координаты текстуры  $(0,0)$ ,  $(63,0)$  и  $(63,63)$  отображаются на вершины треугольника 0, 1 и 2, соответственно. Фактически при этом на треугольник отображается половина текстурной карты. Во втором примере текстура отображается на два смежных треугольника, которые вместе образуют квадрат. В этом случае координаты текстуры выбираются таким образом, что **половина** текстуры отображается на один треугольник, а **остальная часть** — на второй. Здесь текстура прекрасно накладывается на эти два треугольника.

Теперь, когда мы получили визуальное представление о проблеме, ознакомимся с обозначениями на рис. 9.37 и реализуем этот алгоритм в математической форме. Обратите внимание на имена переменных, чтобы легче было следить за кодом,

Интерполируемые величины, **относящиеся** к левой стороне, выражаются следующим образом.

```

dxdyl = (x2 - x0)/(y2 - y0); // Координата x,
                               // интерполируемая для левой стороны
dudyl = (u2 - u0)/(y2 - y0); // Координата u,
                               // интерполируемая для левой стороны
dvdyL = (v2 - v0)/(y2 - y0); // Координата v,
                               // интерполируемая для левой стороны

```

Аналогично выглядят интерполируемые величины, **относящиеся** к правой стороне.

```

dxdyr = (x1 - x0)/(y2 - y0); // Координата x,
                               // интерполируемая для правой стороны
dudyр = (u1 - u0)/(y2 - y0); // Координата u,
                               // интерполируемая для правой стороны
dvdyr = (v1 - v0)/(y2 - y0); // Координата v,
                               // интерполируемая для правой стороны

```

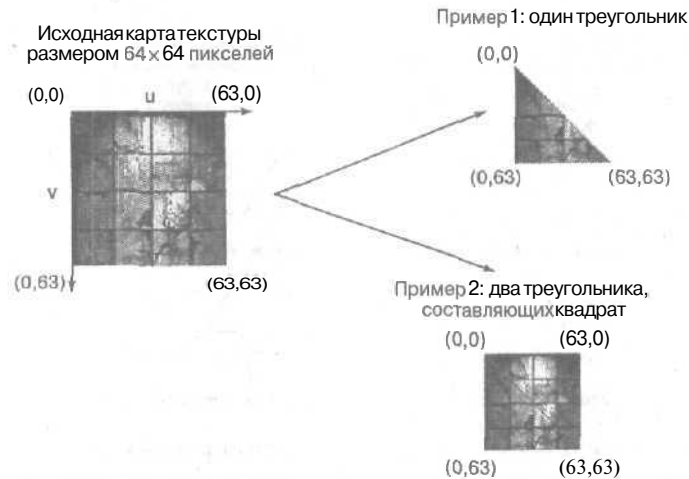


Рис. 9.38. Пример наложения текстуры на треугольник и четырехугольник

Конечно же, приведенные выше математические выражения предоставляют много возможностей для оптимизации. Например, член  $(y2 - y0)$  является общим, и его достаточно вычислить только один раз. Кроме того, лучше вычислить величину, обратную  $(y2 - y0)$ , а затем умножать на нее. Теперь, когда у нас есть **шаги** интерполяции, все почти готово. Чтобы приступить к интерполяции, нужно задать начальные точки. Первой будет самая верхняя вершина 0. В результате начало алгоритма будет выглядеть следующим образом.

```

xl = x0; // Начальная точка для интерполяции x
         // по левой стороне
ul = u0; // Начальная точка для интерполяции u
         // по левой стороне
vl = v0; // Начальная точка для интерполяции v
         // по левой стороне

```

Аналогично представление и для правой стороны.

```

xr = x0; // Начальная точка для интерполяции x
         // по правой стороне
ur = u0; // Начальная точка для интерполяции u
         // по правой стороне
vr = v0; // Начальная точка для интерполяции v
         // по правой стороне

```

Теперь уже все почти готово. Можно приступить к интерполяции вдоль левой и правой сторон треугольника.

```

xl += dxdyt;
ul += dudyt;
vl += dvdyt;

```

```

xr += dxdyr;
ur += dudyr;
vr += dvdyr;

```

Однако это еще не все. Для каждой точки, расположенной на левой и правой сторонах треугольника, нужно выполнить еще одну линейную интерполяцию вдоль строки развертки! Это последняя интерполяция, и именно в ее ходе будут получены координаты текстуры  $(u_i, v_i)$ , которые будут служить индексами строки и столбца битового образа текстуры, необходимыми для получения нужного пикселя. Все, что нужно сделать, — это определить координаты  $u$  и  $v$  точек на левой и правой сторонах, а затем с помощью величины  $dx$  вычислить множитель линейной интерполяции для каждой стороны. Приведем соответствующие математические выражения.

```

dx = (xend - xstart); // Ширина строки развертки
xstart = xl;          // Левая начальная точка
xend = xr;             // Правая начальная точка.

```

Таким образом, интерполируемые величины в пространстве  $u, v$  вдоль каждой строки развертки имеют следующий вид.

```

du = (ul - ur)/dx;
dv = (vl - vr)/dx;

```

Располагая величинами  $du$  и  $dv$ , мы имеем все, что нужно для интерполяции вдоль строки развертки, размещение которой по вертикали задается координатой  $y$ , а по горизонтали — координатами  $xstart$  и  $xend$ . Приведем соответствующий фрагмент кода.

```

// Инициализация интерполируемых величин u и v для левой
// и правой сторон треугольника
ul = ul;
vl = vl;

// Теперь интерполируем слева направо, т.е. в положительном
// направлении оси x
for (x = xstart; x <= xend; x++)
{
    // Извлекаем значение пикселя текстуры
    pixel = texture_map[ul][vl];

    // Выводим пиксель с координатами x,y
    Plot_Pixel(x,y,pixel);

    // Производим сдвиг интерполируемых величин u и v
    ul += du;
    vl += dv;
} //for x

```

Вот и все. Конечно же, во внешнем цикле нужно также выполнять интерполяцию величин  $x_l$ ,  $u_l$ ,  $v_l$ ,  $x_r$ ,  $u_r$ ,  $v_r$  вдоль сторон треугольника, чтобы переходить от одной строки развертки к другой.

При написании функции, производящей отображение текстуры, возникает множество деталей, которые не вошли в приведенные выше псевдофункции. Но, как уже было сказано, функция, предназначенная для растеризации по Гуро, работает так же, как и функция для отображения текстуры. Сначала была разработана именно последняя, КОТО-

рая затем была преобразована в функцию затенения по Гуро. Таким образом, вы уже знакомы с обеими этими функциями. Кроме того, в них совершенно оправданно используются числа с фиксированной точкой. Благодаря этому экономится время, необходимое для преобразования чисел одного типа в другой во внутреннем цикле в процессе вывода пикселей. Этот трюк применяется по той причине, что я не доверяю компилятору. Напомним, что операции с плавающей точкой выполняются с той же скоростью, что и операции с целыми числами. Но код, генерируемый компилятором для операций преобразования типов, выполняется в 10–100 раз дольше, чем математические вычисления.

## Реализация аффинного отображения текстуры

Я думаю, изложенного теоретического материала достаточно для того, чтобы приступить к реализации функций для отображения текстуры. Теперь я хочу обсудить некоторые детали этих функций, а также представить их прототипы и фрагменты кода. Эти функции, как и многие рассмотренные ранее, получились слишком объемными, и привести их код в книгу нет возможности; с ним можно ознакомиться на прилагаемом компакт-диске. (Сожалею, но до тех пор, пока книги не будут выпускаться в электронном виде, представление в них листингов, **занимающих** по 20–30 страниц, сделает их необъятными!) Были реализованы 8- и 16-битовая версии функций, **осуществляющих** отображение текстуры. Единственные представляющие интерес особенности каждой из них содержатся во внутренних циклах, где осуществляется выборка пикселей текстуры и вывод на экран. Если вы помните, в функциях для растеризации по Гуро окончательная интерполяция производится по значениям цветов в формате RGB, из которых потом формируется цвет пикселя. В отличие от них, в функциях, предназначенных для отображения текстуры, окончательные интерполируемые величины и *u* и *v* используются в качестве координат текстуры, а затем производится выборка элементов текстурной карты и вывод пикселя. Билинейные фильтры пока не реализованы.

### Отображение текстуры в 16-битовом режиме

Функция, предназначенная для отображения текстуры в 16-битовом режиме, почти идентична функции для затенения по Гуро. Немногие особенности, присущие этим функциям, содержатся во внутреннем цикле, в котором производится вывод пикселя. Скоро мы вернемся к этому вопросу, а пока что рассмотрим прототип функции и поговорим о том, как она используется.

```
void Draw_Textured_Triangle16(  
    POLYF4DV2_PTR face,    // Указатель на поверхность  
    UCHAR *_dest_buffer,    // Указатель на видеобуфер  
    int mem_pitch)          // Количество байтов в  
                            // строке, 320, 640 и т.д.
```

Мало параметров, вы ожидали большего? Как считает Стивен Вольфрам (Stephen Wolfram), в малом **всегда** содержится большое и всю Вселенную можно описать в нескольких строках кода, а поэтому во многих случаях сложное можно извлечь из простого. Чтобы вызвать эту функцию, достаточно задать значения всех переменных-членов структуры POLYF4DV2, т.е. параметры вершин, карты текстуры, буфера назначения и шага памяти. И на экране **мгновенно** появится изображение треугольника с текстурой, к тому же с учетом отсечения!

Чтобы закрепить навыки применения этой функции, приведем пример ее вызова. Скорее всего, подобный вызов может встретиться в одной из функций визуализации.

```
if (rend_list->poly_ptrs[poly]->attr &  
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
```

```

i
// Задаем параметры вершин
face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->
    tvlist[0].x;
face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->
    tvlist[0].y;
face.tvlist[0].u0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[0].u0;
face.tvlist[0].v0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[0].v0;

face.tvlist[1].x = (int)rend_list->poly_ptrs[poly]->
    tvlist[1].x;
face.tvlist[1].y = (int)rend_list->poly_ptrs[poly]->
    tvlist[1].y;
face.tvlist[1].u0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[1].u0;
face.tvlist[1].v0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[1].v0;

face.tvlist[2].x = (int)rend_list->poly_ptrs[poly]->
    tvlist[2].x;
face.tvlist[2].y = (int)rend_list->poly_ptrs[poly]->
    tvlist[2].y;
face.tvlist[2].u0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[2].u0;
face.tvlist[2].v0 = (int)rend_list->poly_ptrs[poly]->
    tvlist[2].v0;

// Задаем текстуру
face.texture = rend_list->poly_ptrs[poly]->texture;

// Является ли эта текстура обычной излучательной?
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
i
// Выводим треугольник с текстурой как излучательный
Draw_Textured_Triangle16(&face, video_buffer, lpitch);
} // if
else
{
// Выводим его как треугольник с плоским затенением
face.lit_color[0] = rend_list->poly_ptrs[poly]->
    lit_color[0];
Draw_Textured_TriangleFS16(&face, video_buffer, lpitch);
} // else

```

Обратите внимание на участки кода, выделенные полужирным шрифтом. В них задаются параметры треугольника. Как видно из кода, координаты (x, y, u, v) каждой вершины присваиваются соответствующим **переменным-членам** структуры, описывающей поверхность, после чего они используются при вызове функции **Draw\_Textured\_Triangle16()**. Кроме того, нетрудно заметить, что несколькими строками ниже вызывается функция

`Draw_Textured_TriangleFS16()`, представляющая собой версию, поддерживающую плоское затенение, однако к ней мы еще вернемся. На данном этапе функция для отображения текстуры, которая вызывается из функции `Draw_Textured_Triangle16()`, поддерживает лишь постоянное (оно же *излучательное*) затенение. С учетом этого, в модели, на которую будет накладываться текстура, в качестве режима затенения необходимо задавать режим `constant`. Источники, освещающие сцену, в результате не оказывают никакого воздействия на текстуру. Несмотря на то, что скоро это ограничение будет снято, пока что придется смириться с тем, что в первой версии функции, предназначенной для отображения текстуры, поддерживается только постоянное затенение. А теперь ознакомимся с некоторыми важными фрагментами функции `Draw_Textured_Triangle16()`.

Ниже представлен внутренний цикл, содержащийся в этой функции.

```
// Вывод строки развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Вывод пикселя
    screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) + (
        (vi >> FIXP16_SHIFT) << texture_shift2)];
    // Интерполяция величин u и v
    ui += du;
    vi += dv;
} // for xi
```

Рассмотрим строки, выделенные полужирным шрифтом. Во-первых, координаты текстуры `ui` и `vi` заданы в формате `16.16` с фиксированной точкой. Поэтому математическое выражение, с помощью которого производится операция сдвига, выглядит так:

```
screen_ptr[xi] = textmap[ui + vi * word_pitch]
```

Обратите внимание на умножение на `word_pitch`. Конечно же, его рациональнее реализовать как операцию сдвига на величину `texture_shift2`. И поскольку карты текстур представлены в формате со старшей строкой, доступ к ним можно получить как к двумерному массиву с помощью приведенного выше выражения. Единственное, что может представлять интерес, — это извлечение значения `texture_shift2`. Это делается очень просто — достаточно взять двоичный логарифм от ширины текстуры. Таким образом, карте текстуры размером `64x64` (напомним, что она должна быть квадратной) соответствует значение переменной `texture_shift2`, равное 6, поскольку  $2^6 = 64$ , или  $\log_2 64 = 6$ .

Для хранения текстурных карт я использую структуру `BITMAP_IMAGE`. Хотя в этой структуре содержится ширина и высота текстуры, в ней отсутствует степень, в которую нужно возвести двойку, чтобы получить размеры текстуры. Поэтому можно было бы поместить это значение куда-нибудь в структуру, содержащую параметры многоугольника, или в структуру `BITMAP_IMAGE`, в верхние 4 бита какой-нибудь переменной, однако это было бы ненадежно. Вместо этого я решил создать таблицу соответствия для вычисления  $\log_2$ .

```
// logbase2ofx[x] - (int)log2 x, [x:0-256]
UCHAR logbase2ofx[257] =
{
    0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,
    4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
    5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
    5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
    6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
    6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
```

$\geq$ 

Описанный выше способ — **еще** один пример повышения производительности с помощью таблиц соответствия. Что действительно нужно — так это новая структура BITMAP, в которой хранилась бы предварительно вычисленная степень, в которую нужно возвести число 2, чтобы получить размер текстуры. Одна **операция** поиска в таблице соответствия, произведенная для каждого из многоугольников, не слишком замедлит работу программы. Однако следует помнить, что накопление значительного количества таких операций для каждого многоугольника или для каждого пикселя может привести к тому, что программа перестанет быть работоспособной! Нужно постоянно помнить о производительности.

8-битовая версия функции, предназначенной для отображения текстуры с излучательным затенением, очень похожа на ее 16-битовую версию, поскольку в них производятся операции выборки, а не операции с цветами. Следовательно, нет необходимости существенно изменять код. В этой версии операции выполняются не с числами типа **SHORT**, а с числами типа **BYTE**, а в остальном обе версии идентичны и работают одинаково. Таким образом, объем работы значительно сокращается. Прототип 8-битовой версии выглядит следующим образом.

Обновим процессор освещения таким образом, чтобы он поддерживал текстуры. По большей части все уже сделано! Чтобы функция, в которой моделируется освещение, могла поддерживать текстуры, нам буквально не нужно добавлять в нее ни одной

строки кода. Однако некоторые изменения все же необходимы, и сейчас я объясню, какие именно.

В процессе обработки многоугольника системой **освещения** его базовый цвет используется как цвет материала, который затем модулируется цветом освещения. Другими словами, красный многоугольник отражает только красный цвет и т.д. Однако при отображении текстуры мы не смеем даже думать о том, чтобы осветить каждый пиксель текстурной карты, поэтому нужно прибегнуть к трюку. Этот трюк заключается в **следующем**: для каждого многоугольника, на который накладывается текстура, задается белый цвет (255,255,255), а потом его освещение производится как обычно. Однако я написал функцию растеризации, способную обрабатывать многоугольники только с текстурой и плоским затенением, а не с затенением по Гуро, поэтому для многоугольников с текстурой необходимо задавать модель освещения, соответствующую постоянному или плоскому затенению. А теперь вернемся к трюку, о котором я говорил. Как уже было сказано, цвет *каждого* многоугольника, у которого есть текстура, задается белым, затем его освещение моделируется как обычно, а текстура игнорируется.

В результате получается, что на многоугольник воздействует падающий на него свет *всех* цветов, а это именно то, что нужно. Затем на этапе растеризации, при выводе треугольника с текстурой, вместо того, чтобы просто воспринимать карту текстуры в исходном виде, мы модулируем каждый пиксель треугольника интенсивностью, полученной в результате его освещения. Это и есть быстрый способ освещения треугольников с текстурой. Рассмотрим реализацию описанных выше методов в 16- и 8-битовых режимах.

## Добавление освещения АЛЯ визуализации текстуры в 16-битовом режиме

Конечный вид процесса **освещения** и растеризации для 16-битовых текстур с плоским затенением показан на рис. 9.39. Ниже приведен прототип функции, которая выводит многоугольники с текстурой и плоским затенением.

```
void Draw_Textured_TriangleFS16(
    POLYF4DV2_PTR face,      // Указатель на поверхность
    UCHAR *_dest_buffer,     // Указатель на видеобuffer
    int mem_pitch)           // Количество байтов в строке,
                           // 320, 640 и т.д.
```

Этот прототип идентичен прототипу функции без плоского затенения за тем исключением, что к имени этой функции добавлены буквы FS как напоминание о том, что она поддерживает плоское затенение. А теперь рассмотрим некоторые фрагменты функции, чтобы ознакомиться с моделированием освещения в действии. Напомним, что в функции освещения многоугольники обрабатываются так, как будто они белого цвета, а результаты заносятся в элемент массива `lit_color[0]`. Далее нужно просто **промодулировать** цвет текстуры (другими словами, умножить его на интенсивность освещения), в результате чего получим освещенную текстуру. Во-первых, рассмотрим для сравнения старый внутренний цикл, взятый из функции `Draw_Textured_Triangle16()`, которая предназначена для визуализации текстуры с постоянным затенением.

```
// Вывод строки развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Вывод пикселя
    screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT)+
```

```

        ((vi >> FIXP16_SHIFT) << texture_shift2]);
// Интерполяция значений u и v
ui += du;
vi += dv;
} // for xi

```



Рис. 9.39. Конечный вид игрового конвейера с учетом освещения и растеризации текстур

А теперь приведем новый внутренний цикл с модуляцией цвета.

```

// Вывод строки развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Вывод пикселя
    // Сначала извлекаем параметры пикселя
    textel = textmap[((ui >> FIXP16_SHIFT) +
        ((vi >> FIXP16_SHIFT) << texture_shift2));

    // Извлекаем rgb-компоненты
    r_textel = ((textel >> 11) );
    g_textel = ((textel >> 5) S 0x3f);
    b_textel = (textel & 0x1f);

    // Модулируем пиксель текстуры цветом освещенного фона
    r_textel *= r_base;
    g_textel *= g_base;
    b_textel *= b_base;

    // Наконец, выводим пиксель. Заметим, что математические

```

```

// вычисления выполняются так, что в результате мы
// получаем значения г*32, g*64, b*32. Поэтому
// полученные интенсивности каналов нужно разделить на
// 32,64 и 32, соответственно. Однако, поскольку нам
// понадобится выполнять сдвиг результатов, чтобы
// преобразовать их в формат 5.6.5, этого можно
// избежать. Не забывайте, что нужно еще выполнить
// логическое сложение. Пока что мы отложим это до
// оптимизации
screen_ptr[xi] = ((b_textel >> 5) +
                  ((g_textel >> 6) << 5) +
                  ((r_textel >> 5) << 11));

// Интерполяция значений и и v
ui += du;
vi += dv;
} // for xi

```

Из приведенного фрагмента кода видно, что производительность во внутреннем цикле сведена на нет. Для реализации плоского затенения текстуры требуется в 10 раз больше работы, и лишь для того, чтобы добавить плоское затенение. Однако если объектов с текстурой и плоским затенением в игре мало, то функция все же работает. Внутренний цикл устроен так: сначала из текстурной карты нужно извлечь все пиксели текстуры, а затем промодулировать их цветами многоугольника, полученными после его обработки осветительным процессором, и вывести на экран. В результате всех этих манипуляций с битами и операций сдвига производительность сильно падает. На ассемблере все это можно было бы сделать намного лучше; с помощью инструкций, позволяющих одновременно выполнять операции с несколькими величинами, можно добиться четырехкратного повышения производительности. Наконец, некоторых математических операций можно избежать с помощью таблиц соответствия.

На рис. 9.40 приведен снимок экрана, полученный в ходе работы программы DEMO19\_3.CPP|EXE. Это — игровой процессор, выполняющий полное освещение и наложение текстуры с плоским затенением на объект в виде вращающегося куба.

## Добавление освещения для визуализации текстур в 8-битовом режиме

8-битовая версия функции, выполняющей плоское затенение объектов с текстурой, во всех отношениях идентична 16-битовой версии, за исключением внутреннего цикла, в котором происходят необычные вещи. Сначала разберемся с прототипом.

```

void Draw_Textured_TriangleFS(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch)       // Количество байтов в
                        // строке, 320, 640 и т.д.

```

Рассмотрим рис. 9.41, на котором приведена схема использованного алгоритма. Проблема реализации 8-битового освещения заключается в том, что приходится использовать не ступенчатообразное или монохромное, а полнофункциональное освещение с цветами в формате RGB; это аналогично тому, как если бы на скоростной автомобиль поставили тормозное устройство, постоянно замедляющее его скорость. Общая причина, по которой было решено поддерживать 8-битовый режим, заключалась в том, что мы хотели получить быструю программу. А полноценный режим RGB я решил поддерживать

по двум причинам. Во-первых, может возникнуть необходимость создать полнофункциональный игровой процессор, работающий в 8-битовом формате RGB. Во-вторых, операции над RGB-значениями всегда можно свести к монохроматическим вычислениям, однако преобразование из монохроматического формата в формат RGB выполнить намного сложнее. Поэтому было решено реализовать более **общий** случай, а не концентрировать внимание на скорости.

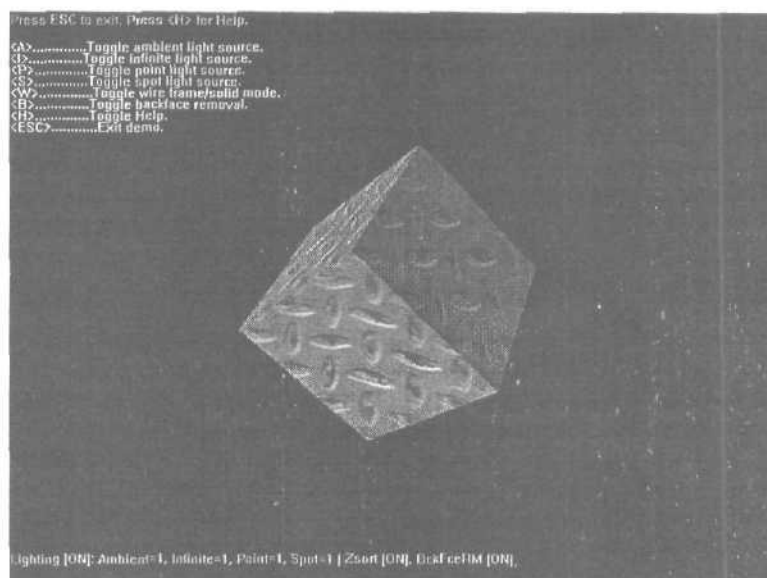


Рис. 9.40. Игровой процессор с освещением и текстурой, работающий в 16-битовом режиме

В любом случае в процессе растеризации многоугольников с плоским затенением и текстурой в 8-битовом режиме возникает проблема. Она заключается в том, что при доступе к 8-битовым картам текстуры нельзя модулировать цвета, поскольку мы находимся в пространстве индексов. Конечно же, можно было бы прибегнуть ко всяким невероятным сложным приемам, например, считывать пиксель текстуры в 8-битовом режиме, находить **соответствующее** RGB-значение в таблице соответствия цветов, модулировать RGB-цвет в элементе массива `lit_color[0]`, выполнять обратное преобразование конечного результата в индекс с помощью соответствующей таблицы **соответствия**, а затем сказать, что программа в 8-битовом режиме работает в два раза медленнее, чем в режиме RGB! Можно также пойти рациональным путем.

Из рис. 9.41 видно, что наша стратегия заключается в создании таблицы соответствия, содержащей всевозможные попарные произведения цветов. Обычно размер такой таблицы равен  $2^{32}$ , т.е. 4 млрд. записей. Однако в 256-цветовом режиме эта таблица может быть компактнее.

В 8-битовом режиме имеется всего 256 цветов, поэтому для полного преобразования в формат RGB достаточно, чтобы таблица содержала 65 536 строк. Таким образом, для всех слов в RGB-формате 5.6.5 мы вычисляем все 256 возможных произведения этого RGB-слова и индекса цвета, который лежит в интервале от 0 до 255, а затем производим поиск в таблице наиболее **подходящего** цвета. Например, нужно **промодулировать** цвет с индексом 26 при помощи RGB-значения (5,10,12) в формате 5.6.5 (т.е. значениями в ингер-

валах  $(0..31, 0..63, 0..31)$ . Таким образом, в формате 8.8.8 нужно выполнить умножение на  $(8, 4, 8) \cdot (5, 10, 12) = (40, 40, 96)$ .

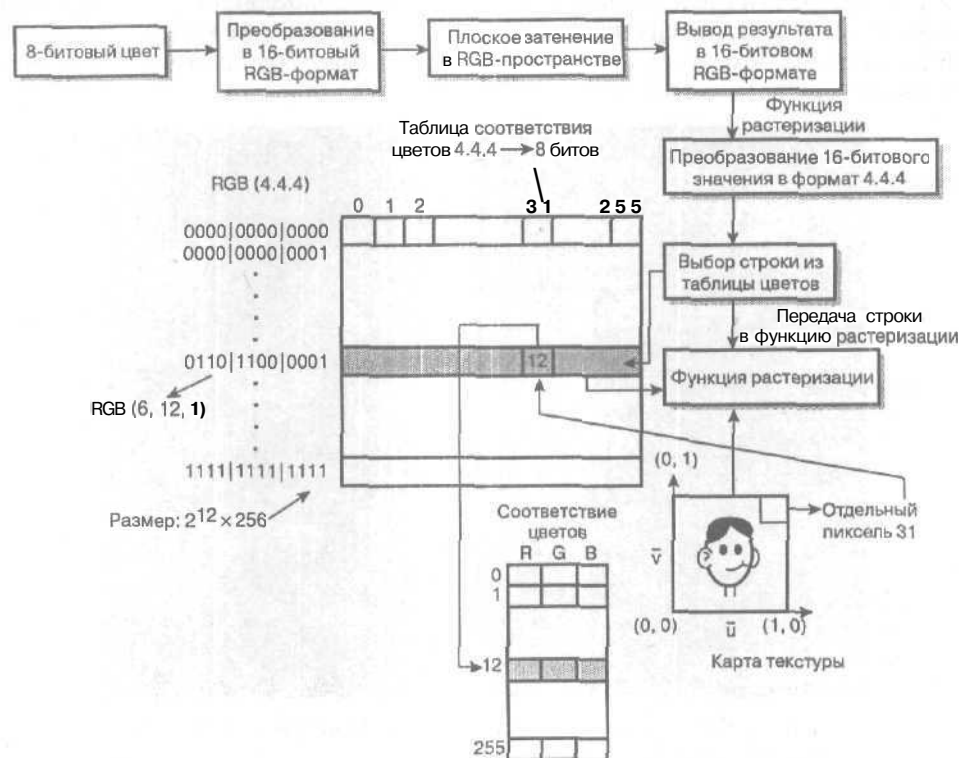


Рис. 9.41. Схема обработки данных в 8-битовом процессоре освещения

Для этого сначала извлекается RGB-значение, соответствующее индексу 26. Пусть это будет значение  $(90, 12, 60)$ . Напомним, что мы работаем в режиме 8.8.8. Затем RGB-значение, проиндексированное номером 26, умножается на значение  $(40, 40, 96)$  в формате 8.8.8. В результате получим:

конечный цвет -  $(90, 12, 60) * (40, 40, 96) =$   
 $(3600, 480, 5760)$

Далее, этот результат нужно преобразовать таким образом, чтобы он находился в интервале от 0 до 255. Для этого нужно поделить все компоненты на 255, в результате чего получим тройку таких целых чисел:

$(14, 1, 22)$

Что же все это означает? Приведенное выше RGB-значение — это цвет в формате RGB, полученный в результате модулирования цвета с индексом 26 цветом  $(5, 10, 12)$ , заданным в RGB-формате 5.6.5. Таким образом, мы описали шаг, состоящий в модуляции цветового индекса одним из возможных RGB-значений. Чтобы завершить этот шаг, нужно просмотреть таблицу соответствия цветов, найти индекс, соответствующий значению  $(5, 10, 12)$ , и занести его в память. Для поиска используется массив

`color_lookup[RGB_color][color_index]`

в котором значения индекса `RGBcolor` изменяются в интервале 0..65535, а значения индекса `color_index` — в интервале 0..255. Получается огромная таблица объемом 16.7 Мбайт. Неужели необходимы все ее элементы? Ответ — нет! Можно применить трюк, состоящий в предварительном преобразовании всех RGB-значений в формат 4.4.4, в котором красный, зеленый и синий цвета представлены 16 оттенками, а затем использовать их как модулирующие величины для всех 256 цветов таблицы. Такая таблица заняла бы всего  $2^{12} \cdot 256 = 1$  Мбайт, что вполне разумно. Единственная проблема состоит в том, чтобы создать такую таблицу. На процессоре Pentium IV этот процесс занимает около 20 секунд! Такая таблица создается с помощью следующей функции.

```
int RGB_12_8_Lighting_Table_Builder(
    LPPALETTEENTRYsrc_palette,
    UCHARrgblookup[4096][256])
{
    // Эта функция создает таблицу цветов, используемую для
    // освещения в 8-битовом режиме в функции,
    // предназначенной для отображения текстуры. Нам нужна
    // таблица, которая всем возможным произведениям цвета
    // освещения и цвета текстуры сопоставляет наиболее
    // подходящий цвет. Однако такая таблица состояла бы из
    // 256*65536 записей, а это многовато. Поэтому входное
    // значение освещения преобразуется в формат 4.4.4, в
    // котором каждый канал представлен 16 различными
    // интенсивностями, что составляет в сумме 12 битов. В
    // этом случае получится таблица объемом 256*2^12 = 1
    // Мб, что вполне разумно и дает выигрыш в
    // производительности. Таблица будет двумерной. Строки в
    // ней будут соответствовать различным интенсивностям, а
    // столбцы - цветам. Этот формат более эффективен, так
    // как таблицы хранятся в формате со старшей строкой.
    // Поскольку выполняется плоское затенение с текстурой,
    // то известно, что в ходе вычислений изменяется цвет
    // текстуры, но не цвет освещения. Таким образом,
    // когерентность кэша будет превосходной, если доступ к
    // элементам производится в виде:
    // rgblookup[RGBcolor.12bit][textel index.8bit]

    // Сначала проверяем указатели
    if(!src_palette || !rgblookup)
        return (-1);

    // Всего нужно вычислить 4096 RGB-значений, если они
    // заданы в RGB-формате 4.4.4
    for (int rgbindex=0; rgbindex< 4096; rgbindex++)
    {
        // Нужно перемножить каждый цвет в интервале 0..4095
        // и формате RGB на каждый из 255 цветов палитры, а
        // затем найти среди полученных цветов наиболее
        // подходящий
        for (int color_index=0; color_index < 256;
            color_index++)
        {
            int curr_index = -1;    // Текущий индекс
                                   // цвета - кандидата
```

```

        // для наиболее
        // подходящего цвета
Long curr_error = INT_MAX; // Расстояние до
        // наиболее
        // подходящего цвета
        // или "отклонение"

// Извлечение из rgb-индекса интенсивностей
// каналов rgb в предположении, что цвет задан
// в формате 4.4.4
int r = (rgbindex >> 8);
int g = ((rgbindex >> 4) & 0x0f);
int b = (rgbindex & 0x0f);

// В конечном итоге эти значения каналов r,g и
// b, имитирующих источник освещения, нужно
// умножить на цвет пикселя текстуры. Выполняем
// модулирование, проверяем, что результаты
// находятся в интервале 0..255, и делим
// результаты на 15, поскольку значения r,g,b
// нормированы не на 1.0, а на 15
r = (int)((float)r *
    (float)src_palette[color_index].peRed) / 15);
g = (int)((float)g *
    (float)src_palette[color_index].peGreen) / 15);
b = (int)((float)b *
    (float)src_palette[color_index].peBlue) / 15);

// Производим поиск этого цвета в палитре
for (int color_scan = 0; color_scan < 256;
    color_scan++)
{
    // Вычисляем расстояние между исходным
    // и целевым цветами
    long delta_red =
        abs(src_palette[color_scan].peRed - r);
    long delta_green =
        abs(src_palette[color_scan].peGreen - g);
    long delta_blue =
        abs(src_palette[color_scan].peBlue - b);
    Long error = (delta_red*delta_red) +
        (delta_green*delta_green) +
        (delta_blue*delta_blue);

    // Подходит ли цвет лучше остальных?
    if (error < curr_error)
    {
        curr_index = color_scan;
        curr_error = error;
    } // if
} // for color_scan

// Найден наиболее подходящий цвет, вводим его в
// таблицу
rgblookup[rgbindex][color_index] = curr_index;
} // for color_index

```

```

    } // for rgbindex
    // Код успешного выполнения
    return(l);
} // RGB_12_8_Lighting_Table_Builder

```

Для хранения таблицы была создана глобальная переменная

```
UCHAR rgbLightlookup[4096][256];
```

Таким образом, при работе в 8-битовом режиме таблицу нужно создавать после загрузки палитры, которая производится из библиотеки `T3DLIB1.CPP` в переменную палитры.

А теперь вернемся к самой функции, производящей растеризацию объектов с плоским затенением в 8-битовом режиме. Дело вот в чем: таблица, которую мы создали, состоит из 4096 строк, в каждой из которых по 256 значений. Каждая строка таблицы представляет значение цвета в формате RGB 4.4.4, умноженное на каждый из 256 цветов, а также цвета, которые лучше всего подходят к элементам данной строки. Поэтому все, что мы делаем при входе в эту функцию, — получаем указатель на строку, которая соответствует цвету освещения многоугольника. Во внутреннем цикле функции растеризации мы работаем с одномерной таблицей, содержащую индексы цветов, из которых состоит карта текстуры. Фактически, эта таблица выполняет операцию модулирования освещения. Приведем соответствующий фрагмент кода.

```

// Вывод строки развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Вывод пикселя текстуры
    // Извлечение пикселя текстуры
    textel = textmap[(ui >> FIXP16_SHIFT) +
        ((vi >> FIXP16_SHIFT) << texture_shift2)];
    // Модулирование базовым цветом с помощью таблицы
    // освещения и запись пикселя текстуры
    screen_ptr[xi] = lightrow444_8[textel];
    // Интерполяция величин и и v
    ui += du;
    vi += dv;
} // for xi

```

Обратите внимание на обращение к массиву `lightrow444_8[]`. Это обращение к строке, в которой все 256 цветов "освещены" (т.е. промодулированы) входящим белым цветом многоугольника, на который накладывается текстура. Аналогичный метод можно применить и при освещении в 16-битовом режиме! Таблицы получатся огромными, однако при разумном подходе можно добиться того, чтобы внутренний цикл 16-битовой версии программы выглядел аналогично описанному выше. В крайнем случае, одну таблицу соответствия можно представить в виде двух!

В качестве демонстрационного примера ознакомьтесь с работой 8-битовой версии программы `DEMO119_3_8b.CPP EXE`, выполняющей отображение текстуры с плоским затенением. В этом процессоре производится полноцветное освещение вращающегося куба.

## Вопросы оптимизации для 8- и 16-битовых режимов

В этой главе было упомянуто много возможностей и идей по оптимизации. Подавляющее большинство из них не было реализовано, поскольку это настолько усложнило

бы код, что его было бы трудно воспринимать. Тем не менее, освежим в памяти некоторые идеи, чтобы вы не забывали о них.

## Таблицы соответствия

Таблицы соответствия пригодны почти везде. Пока этот вопрос был затронут лишь поверхностно. Присмотревшись к любой функции, реализующей какой-либо алгоритм, можно заметить, что все входные ее параметры можно объединить в виде одного индекса объемом несколько килобайтов, а затем создать таблицу, ставящую в соответствие входным параметрам выходное значение (или набор выходных значений). В большинстве случаев объем такой таблицы не превысил бы нескольких мегабайтов, а современные компьютеры имеют в своем распоряжении значительное количество памяти. Тот, на котором работаю я, оснащен 2.0Гбайт оперативной памяти, поэтому даже если в вашей игре таблицы соответствия займут 32–64 Мбайт — это нормально.

Понятно, что таблицы соответствия в основном используются для математических преобразований, однако при работе с трехмерной графикой их применение значительно расширяется. Это может быть отображение цветов, отображение текстур и другие подобные операции. Кроме того, заслуживают внимания так называемые *каскадные таблицы соответствия* (cascading lookup tables). Поясним, что это такое.

Предположим, у нас заданы две входные величины,  $x$  и  $y$ , каждая из которых занимает по 16 битов памяти, и для всевозможных комбинаций этих величин нужно вывести однобайтовое значение. Количество комбинаций для входных значений составляет два в степени  $16+16=32$  или 4.2 млрд. байтов. Такая таблица соответствия займет слишком много места. Однако задачу можно разбить на этапы, например, сначала обработать по 8 старших битов каждого операнда, найти для них соответствующее значение (необходимая для этого таблица имеет размер два в степени  $(8+8)=16$ , или 64 Кбайт), и на его основе осуществить еще один поиск по оставшимся 16 битам.

Предположим, выходное значение первой таблицы займет 8 битов. Прибавив длину остальных 16-битовых данных, получим 24 бита, для которых нужно найти соответствующее значение длиной в 1 байт. Такая таблица займет всего 16 млн. байтов. Это все еще много, но намного меньше, чем 4 млрд.! Суть в том, что путем разбиения одной задачи на несколько более мелких иногда можно добиться снижения объема поискового пространства, причем это снижение будет происходить по логарифмическому закону. Таким образом, разбивая входные параметры на две половины или на более мелкие части, можно свести процесс к нескольким этапам и при этом существенно уменьшить количество используемого пространства.

Кроме того, не бойтесь выполнять масштабное преобразование входных данных, чтобы с ними было проще работать. Например, в функции, предназначенной для отображения текстуры с плоским затенением в 8-битовом режиме, для поиска индекса цвета, который лучше всего подходит цветам, заданным индексами в формате 8.8.8, использована таблица соответствия, работающая в режиме RGB 4.4.4. Стоимость этой таблицы, выраженная в объеме занимаемой ею памяти, незначительна, однако благодаря ей процесс сильно ускоряется. На самом деле мы работаем в пространстве цветов в формате 5.6.5, поэтому в результате преобразования к формату 4.4.4 теряется половина пространства цветов. Но в 8-битовом режиме этого никто не заметит, поэтому над трюками подобного рода стоит поразмышлять.

## Связность вершин каркаса

Несмотря на то, что в коде, производящем затенение вершин при освещении по Гуро, не реализованы никакие методы, позволяющие восстановить связь между вершинами каркаса при разбиении его на многоугольники, эта оптимизация представляется весьма

целесообразной. Зачем же освещать 36 вершин куба, если достаточно выполнить эту операцию только для восьми? Напомним, что пока объект остается цельным, между его вершинами и многоугольниками существует взаимно однозначное соответствие и нет никаких дублирований. Это справедливо для нормалей в вершинах, координат текстур и других параметров. Поэтому если есть возможность оптимизации при сохранении объекта целым (например, при его освещении), это нужно делать незамедлительно. Конечно же, не хочется освещать многоугольники, которые находятся на обратной стороне объекта и т.п., поэтому следует быть особенно внимательным и осторожным.

## Кэширование математических результатов

С этим приемом мы уже немного сталкивались. Это было при рассмотрении новой функции освещения, когда однажды вычисленные нормали к многоугольникам кэшировались для их дальнейшего использования в функциях, предназначенных для моделирования точечных источников и световых пятен. Конечно же, есть много других возможностей для кэширования такого рода. Достаточно просто просмотреть код и отметить, какие величины вычисляются по несколько раз. Это и будут кандидаты в кэшируемые величины. Метод работы с подобными переменными заключается в том, что на начальном этапе им присваивается "метка", соответствующая бесконечно большой величине. Перед тем как использовать данные, проверяется значение содержащей их переменной. Если оно равно бесконечности, то необходимое значение вычисляется и присваивается переменной. В противном случае этап вычислений опускается и происходит переход к следующему фрагменту кода. Например, приведем версию некоторой функции без кэширования.

```
// Блок кода
f1 = sin(x) * cos(y) + zeta
ft = f1 * cos(y)
f3 = alpha + beta * sin(x)
```

А теперь предположим, что величины  $x$ ,  $zeta$ ,  $\alpha$  и  $\beta$  изменяются в цикле по какому-то своим законам, а величина  $y$  остается постоянной. Таким образом, значение  $\cos(y)$  можно занести в память. Это делается следующим образом.

```
cosy = cos(y)

// Блок кода
f1 = sin(x) * cosy + zeta
ft = f1 * cosy
f3 = alpha + beta * sin(x)
```

Однако в таком фрагменте возникает проблема. Дело в том, что мы заранее знаем, что значение  $\cos(y)$  понадобится в последующих строках кода, поэтому и вычисляем его. Однако если бы это было неизвестно, то не хотелось бы тратить время на вычисление этой величины до тех пор, пока она не понадобится. Например, может случиться так, что обращение к приведенному блоку кода вообще не производится, и время, которое понадобилось для вычисления  $\cos(y)$ , потрачено напрасно. Поэтому более разумная стратегия заключается в том, чтобы переменной, которая может использоваться, а может и не использоваться, присвоить значение бесконечность (или какое-нибудь другое недопустимое значение-метку) и вычислить эту переменную лишь в том случае, если она понадобится.

```
// Инициализация
float cosy = FLT_MAX;

// Некоторый код
```

```
// Этот фрагмент кода вычисляется только  
// если понадобится использовать cosy  
if (cosy == FLT_MAX)  
    cosy = cos(y);
```

## Использование возможностей SIMD

**SIMD** (single instruction multiple data) переводится как "архитектура с одним потоком инструкций и несколькими потоками данных". Это часть архитектуры процессора Pentium ПК, позволяющая выполнять одну и ту же операцию с несколькими потоками данных. Другими словами, можно перемножить четыре числа на другие четыре числа и т.д. Такая возможность идеально подходит для операций, которые приходится выполнять при обработке трехмерной графики. Вы можете спросить; "Это то же самое, что и MMX?". И да, и нет. MMX (multimedia extensions — мультимедийные расширения) представляют первое поколение этой технологии, причем они не были доработаны. Они предоставили процессору Pentium возможность одновременно выполнять четыре операции с целыми числами, но это достигалось за счет использования регистров, предназначенных для работы с числами с плавающей точкой. По поводу особенности SIMD следует заметить, что она отлично работает!

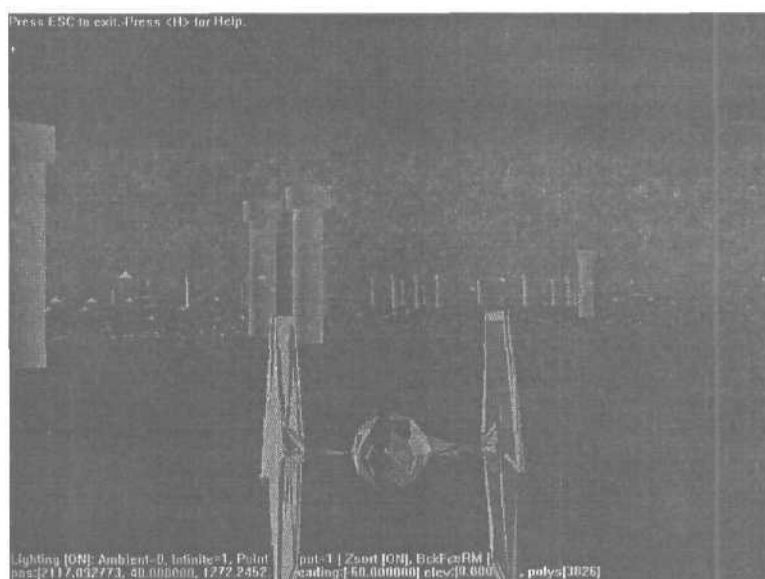
Например, на процессоре Pentium 4 с помощью встроенных инструкций SIMD и (или) внутренних функций, лучше всего использующих особенности данного компьютера, можно ускорить выполнение математических операций и алгоритмов трехмерной графики почти на 400%. В конце книги кратко будет описано, как это делается, и здесь определенно есть на что посмотреть. Представьте себе, насколько быстрее можно выполнить освещение, затенение по Гуро, наложение текстуры и преобразование многоугольников, и это далеко не полный список! Чтобы ознакомиться с инструкциями SIMD, посетите узел Internet по адресу <http://www.intel.com> и произведите поиск по ключевым словам sse, sse2 и simd. Кроме того, по адресу [msdn.microsoft.com/vstudio/downloads/ppack/default.asp](http://msdn.microsoft.com/vstudio/downloads/ppack/default.asp) можно загрузить пакет программ для компилятора Visual C++, позволяющий ему поддерживать инструкции SSE (streaming SIMD extensions — потоковые расширения SIMD)/SSE2 (streaming SIMD extensions 2 — потоковые расширения SIMD 2).

## Итоговые демонстрационные программы

Пришло время ознакомиться с программами, демонстрирующими работу игрового процессора. Я уже начинаю сходить с ума от этого танка, однако решил подождать, пока мы не получим в свое распоряжение технологию, позволяющую сделать что-нибудь стоящее. Чтобы получить такую замечательную программу, пришлось потратить около 20 часов. Однако не составляет особого труда обновить ее всеми дополнительными возможностями, в результате чего получена 16-битовая версия DEMOII9\_4.CPP|EXE и 8-битовая версия DEMOII9\_4\_8b.CPP|EXE. Обратите внимание на высококачественное затенение изображения на рис. 9.42, на котором приведена копия экрана, полученная при работе этой программы. По сути, это тот же пример, с которым мы уже знакомы, но в нем для многих объектов добавлено затенение по Гуро, а на движущийся объект, которым можно управлять, наложена текстура с плоским затенением. Как обычно, работой программы можно управлять с помощью клавиш, перечисленных в табл. 9.1.

**Таблица 9.1. Управляющие клавиши программы DEMOII9\_4xx.CPP I EXE**

Клавиша	Функция
<A>	Переключение источника общего освещения
<I>	Переключение бесконечно удаленного источника
<P>	Переключение точечного источника
<S>	Переключение источника в виде светового пятна
<W>	Переключение каркасного/заполненного режима
<B>	Включение и выключение удаления обратных поверхностей
<L>	Переключение процессора освещения
Стрелка вправо	Поворот вправо
Стрелка влево	Поворот влево
Стрелка вверх	Движение вперед
Стрелка вниз	Движение назад
Пробел	Турборежим
<H>	Вызов справки
<Esc>	Выход из программы



*Рис. 9.42. Демонстрационная программа танкового симулятора с затенением по Гуро и отображением текстуры*

## Raiders 3D II

А теперь немного позабавимся! Я решил обновить демонстрационную программу Raiders 3D, с которой мы познакомились в главе 1, "Основы программирования трехмерных игр", и сделал ее полнофункциональную трехмерную версию со сплошными

объектами. Снимок экрана, полученный в результате работы этой программы, приведен на рис. 9.43. Программа поддерживает все рассмотренные к **настоящему** моменту возможности: цветные источники освещения, отображение текстуры и многое другое. Это базовая **демонстрационная** программа, в которую можно добавлять все, что угодно. Имена исполняемых файлов программы — `RAIDERS3D_2.CPP|EXE` и `RAIDERS3D_2B.CPP|EXE` (с немного **отличающейся** цветовой схемой). Для компиляции программы понадобится скомпоновать ее с библиотечными модулями от `T3DLIB1.CPP|H` до `T3DLIB7.CPP|H`. Кроме того, не забудьте скопировать в рабочий каталог все необходимые файлы. Напомним также, что для всех карт текстур используется каталог с именем

```
char texture_path[80] = "./";
```

Возможно, вы захотите задать имя `"TEXTURE/"` или какое-нибудь другое, однако далее нужно помнить, что все текстуры должны находиться в этом каталоге, а сам он (в свою очередь) ~ в рабочем каталоге, содержащем исполняемый файл с расширением `.EXE`.

Эта версия игры подобна той, которая приведена в главе 1, "Основы программирования трехмерных игр", однако она более совершенна. Прицел перемещается с **помощью** мыши, а чтобы открыть огонь, следует воспользоваться левой кнопкой мыши. Чтобы выйти из игры, нажмите клавишу `<Esc>`. Логика игры очень проста. Необходимо уничтожить всех приближающихся врагов с помощью лучевого оружия. Игра заканчивается после того, как от вас спасутся 25 врагов. Чтобы запустить ее заново, нажмите клавишу `<Enter>`. Есть также другие управляющие клавиши, например:

`<W>` — переключение каркасного режима/режима заполненного объекта;

`<I>` — переключение первичного точечного источника освещения, имитирующего Солнце;

`<A>` — переключение источника общего освещения.

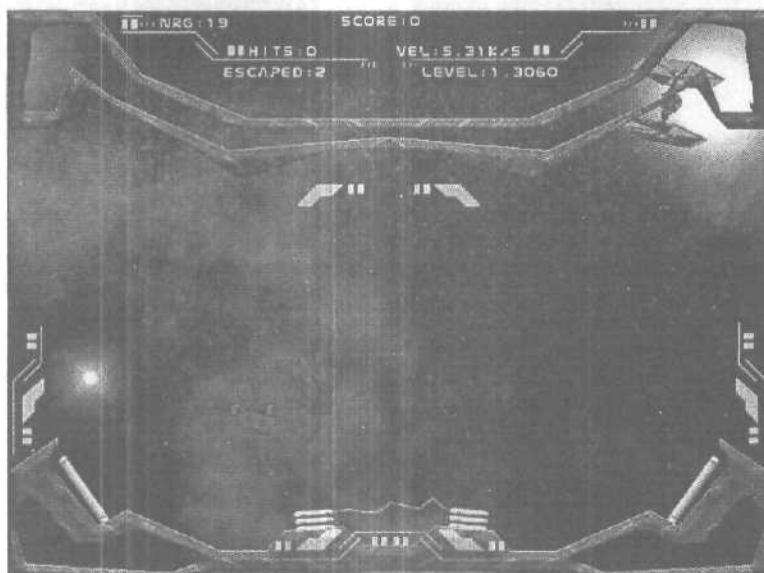


Рис. 9.43. *Raiders 3D II*

Несмотря на простоту игры, в ней имеется несколько интересных особенностей, которые я хотел бы обсудить, чтобы в дальнейшем вы **могли** добавлять их в свои игры.

## Имитация звезд

Трехмерное звездное небо реализуется весьма хитрым способом. Наш игровой процессор не поддерживает звезды, поэтому они включены в игру и визуализируются вручную, с тем же расстоянием обзора и размером окна, что и для прочих объектов. Однако по большей части логика визуализации звезд совпадает с той, которая применялась в демонстрационной программе `RAIDERS3D.CPP|EXE`, встречавшейся нам в главе 1. Кстати, даже странно, что до сих пор нами не реализована поддержка звезд в виде маленьких частиц, ведь их не нужно ни освещать, ни сортировать по оси  $z$ . Однако не вызывает сомнений, что вскоре в игровой процессор понадобится добавить такую возможность, чтобы не приходилось прибегать к различным трюкам. С другой стороны, я терпеть не могу насилие внедрять в процесс визуализации в игровом процессоре простые вещи, подобные поддержке звезд в виде маленьких частиц, поэтому здесь нужно подумать над тем, как достичь золотой середины.

## Энергетическое оружие

Это просто замечательный эффект. Сначала оружие было представлено обычными прямыми линиями, но потом я подумал, что было бы здорово имитировать энергию сходящегося пучка частиц или плазмы. Чтобы реализовать такое оружие, я просто написал код, обозначающий две конечные точки, между которыми будет выводиться энергетический луч, несколько сегментов, на которые этот луч разбивается, и наконец, его максимальную амплитуду. Удивительно, что это сработало с первой попытки и при этом все выглядит замечательно!

## Выявление столкновений и слежение за целью

Чтобы прицел работал как следует, пришлось немного потрудиться. По сути, в игре предусмотрена прицельная сетка, которую можно перемещать в двумерном пространстве экрана. Однако объекты движутся не в двумерном, а в трехмерном пространстве. Поэтому приходится применять трюк, состоящий в том, что условная прямоугольная оболочка, окружающая каждый звездолет, проектируется на плоскость экрана, а затем производится проверка, находится ли центр прицела в рамках этой проекции. Чтобы реализовать эту задачу, были определены параметры прямоугольной оболочки вокруг каждого звездолета, а затем выполнялось проецирование четырех вершин этой оболочки. После этого производилась проверка, попадает ли сетка прицела на эту проекцию. Как видите, не было использовано ничего такого, с чем вы бы уже не сталкивались, просто здесь продемонстрировано другое применение уже известных приемов. Конечно же, важно уметь применять этот метод на практике, поскольку он составляет основу реализации двумерного прицела во всех подобных трехмерных играх. Помните одно: сначала следует спроецировать на плоскость экрана прямоугольную рамку, ограничивающую трехмерное изображение, а затем производить с ней необходимые действия.

## Взрывы

Взрывы оказались не такими драматичными, как мне бы хотелось. Суть этого алгоритма состоит в следующем. Обычно берется набор многоугольников, из которых состоит инопланетный звездолет, и копируется в имеющуюся сетку взрыва. Однако я поступил несколько по-другому. Я преднамеренно отказался от связности вершин, скопировал один за другим каждый многоугольник и воспроизвел в каждом из них информацию о вершинах. Это было сделано таким образом, что во время взрыва каждый объект можно было преобразовывать или анимировать независимо от других. Далее, чтобы имитировать взрыв, я разбросал многоугольники (осколки звездолета) под разными углами и

с разными скоростями. Получилось неплохо, однако это не идет ни в какое сравнение со взрывом, моделируемым с **помощью** битовых образов! Вы вполне можете добавить его поверх **разлетающихся** многоугольников. Игра от этого станет лучше. У меня не хватило для этого времени, но вам это будет полезно.

## Процессор шрифтов

Как известно, GDI (Graphic Device Interface — интерфейс графических устройств) работает очень медленно. При выводе звезд или отладочной информации его скорости хватает, но для отображения текстовых сообщений в ходе игры она явно мала. Поэтому мне пришлось написать для этой игры простенький **процессор шрифтов**. В нем используются две **функции**, которые можно найти не в библиотечных модулях, а в самом исходном файле RAIDERS3D\_2.CPP. Прототип первой функции выглядит следующим образом.

```
int Load_Bitmap_Font(char *fontfile, BOB_PTR font);
```

Эта функция загружает битовый образ с **шаблонным** шрифтом. На этот битовый образ накладывается ряд требований. Шрифт должен содержать ровно 64 символа и размещаться в 4 строки по 16 символов в каждой. Кроме того, размер каждого символа должен быть равен 16x14 пикселей. В качестве первого символа должен быть задан пробел. Ознакомьтесь с файлом **tech\_char\_set\_01.bmp**, который содержится в папке, соответствующей настоящей главе. Кроме того, понадобится загрузить шрифт в незанятый объект **блиттера**, который разместит его в нужном месте.

Следующая функция используется для отображения на экране надписи так, чтобы она начиналась в точке с координатами (x,y). В функцию передается объект **блиттера**, содержащий битовый образ шрифта, начальные координаты x и y, содержимое, высоту и ширину надписи, а также поверхность, на которой эта надпись выводится. Ниже представлен прототип функции.

```
int Draw_Bitmap_Font_String(BOB_PTR font
    intx, inty,
    char *string,
    int hpitch, int vpitch,
    LPDIRECTDRAWSURFACE7 dest);
```

Использовать функцию очень просто. Предлагаю ознакомиться с ее **кодом**, чтобы понять, как она работает. Возможно, в следующей главе ее доработанная версия будет добавлена в нашу библиотеку.

## Освещение в игре

Вся эта глава была посвящена моделированию освещения! Рассматриваемый небольшой пример тоже имеет отношение к этой теме. В процессоре этой **игры** есть источники четырех видов:

- источники общего освещения;
- имитация солнечного (белого) света в виде точечного источника, размещенного в левом нижнем углу возле битового изображения Солнца;
- имитация солнечного (красного) света в виде точечного источника, размещенного в левом нижнем углу возле битового изображения Солнца;
- разряды энергии, представляющие собой точечные источники, размещенные на протяжении нескольких сотен единиц впереди звездолета; они активизируются только во время стрельбы из оружия,

## Заключительные комментарии

В игре лишь в небольшой степени используются возможности нашего небольшого игрового **процессора**. Поработав немного над этой демонстрационной версией, возможно, удастся превратить ее в полнофункциональную трехмерную игру для продажи! Для этого в нее необходимо добавить уровни, поставить на каждом из них **свою** цель, положить в основу игры сюжетную линию и т.д., а технология уже есть. Кроме того, помните, что производительность этой игры составляет всего десятую или двадцатую долю от той, которую можно получить после оптимизации. Не забудьте испытать как красную (RAIDERS3D\_2.EXE), так и зеленую (RAIDERS3D\_2B.EXE) версии этой игры. Они отличаются лишь цветами фона.

## Резюме

Если даже вы решите прекратить чтение книги на этом месте, то все равно будете в состоянии создать трехмерный игровой процессор с полноценным освещением, цветами, наложением текстуры и поддержкой 8-битового и 16-битового режимов. Прошу вас хорошо разобраться в коде, представленном в данной главе, а также поэкспериментировать с демонстрационными примерами. У меня не было времени, чтобы доработать процессор и создать действительно впечатляющие примеры, и я лишь в небольшой мере использовал возможности нашего "сырого", неоптимизированного процессора. Уверен, что вы сможете добиться при работе с ним удивительных результатов!



# ГЛАВА 10

## Отсечение в трехмерном пространстве

### В этой главе...

• Общее представление об отсечении	880
• Теоретические основы алгоритмов отсечения	884
• Практическое отсечение по границам области обзора	900
• Игры с ландшафтами	935

Довольно долго нам удавалось избегать отсечения в трехмерном пространстве, но наконец назрела необходимость реализовать и эту процедуру. Уклоняться от этого вопроса далее просто невозможно. В данной главе излагается как теоретический, так и практический материал по трехмерному **отсечению**, приводится обоснование **применяющихся** в этой области методов, а также рассматриваются различные действительно эффективно работающие технологии. Вот о чем идет речь в данной главе:

- общее представление об отсечении;
- теория алгоритмов отсечения;
- **реализация** отсечения по границам области обзора;
- моделирование ландшафта.

## Общее представление об отсечении

В общем плане отсечение многократно обсуждалось как в этой книге, так и в книге *Программирование игр для Windows. Советы профессионала*. В трехмерной графике эта процедура играет неоценимую роль, поскольку геометрические объекты без **надлежащего** отсечения не только выводятся на экран неправильно, но и вызывают ситуации деления на **ноль**, сбои, связанные с защитой памяти, и многие другие неприятные проблемы. Рассмотрим вкратце различные виды отсечения, а также причины использования каждого из них.

### Отсечение в пространстве объекта

Отсечение в пространстве объекта выполняется на уровне элементарных составляющих в определенной **геометрической** области. Область отсечения может быть двумерной или **трехмерной**, однако суть в том, что отсечение в пространстве объекта производится в математическом пространстве с помощью математических представлений объектов, многоугольников, геометрических примитивов и **других** элементов. Приятной особенностью этого вида отсечения является его абстрактность и доступность для понимания: имеется объект или список многоугольников, и мы производим отсечение по границам двумерной или трехмерной области обзора. После этого обработанные многоугольники (в нашем случае треугольники) поступают на следующий этап работы трехмерного конвейера визуализации.

Проблему при этом подходе вызывают детали реализации. В нашем игровом процессоре всегда применяются треугольники, отсечение которых выполняется либо по границам двумерного прямоугольника, либо по границам трехмерной призмы, которыми ограничивается видимое пространство. При этом получаются многоугольники, количество вершин которых больше трех (т.е. отличающиеся от треугольников). Другими словами, в результате отсечения по **границам какой-либо** области каждый треугольник может превратиться в четырехугольник или многоугольник с большим количеством вершин. После этого такие многоугольники нужно снова разбить на треугольники, и не всегда понятно, как это делать. Если бы наш игровой процессор мог работать с произвольными многоугольниками, такая **операция** не потребовалась бы, однако в нем уже проведены различные оптимизации, основанные именно на том, что мы используем только треугольники. Вторая проблема связана с тем, что процесс отсечения в любом пространстве сводится к нахождению областей пересечения линий с линиями, линий с плоскостями и т.д. Возможно, на первый взгляд и кажется, что здесь все просто, однако на самом деле это далеко не так, и такая операция приводит к многочисленным проблемам.

### Двумерное отсечение в пространстве объекта

С отсечением в двумерном пространстве мы уже сталкивались. В его основе лежит представление о том, что имеются линии или треугольники, **уже спроецированные** на отображаемую плоскость прямоугольной формы, и нужно отсечь все, что выходит за рамки этой плоскости. Код, с помощью которого это делается, описан как в этой книге, так и в предыдущей (причем там это сделано более подробно). Проблема в том, что методы двумерного отсечения нам больше не помогут, потому что наша основная проблема заключается в том, что в трехмерном пространстве проецируемые многоугольники могут быть весьма протяженными, такими, что не исключена возможность их выхода не только за ближнюю плоскость отсечения, но и за пределы плоскости  $z = 0$ . В этом случае спроецированные вершины имели бы отрицательную координату  $z$ . Подобная ситуация показана на рис. 10.1. Таким образом, в трехмерном пространстве понадобится более интенсивное отсечение в пространстве объекта.

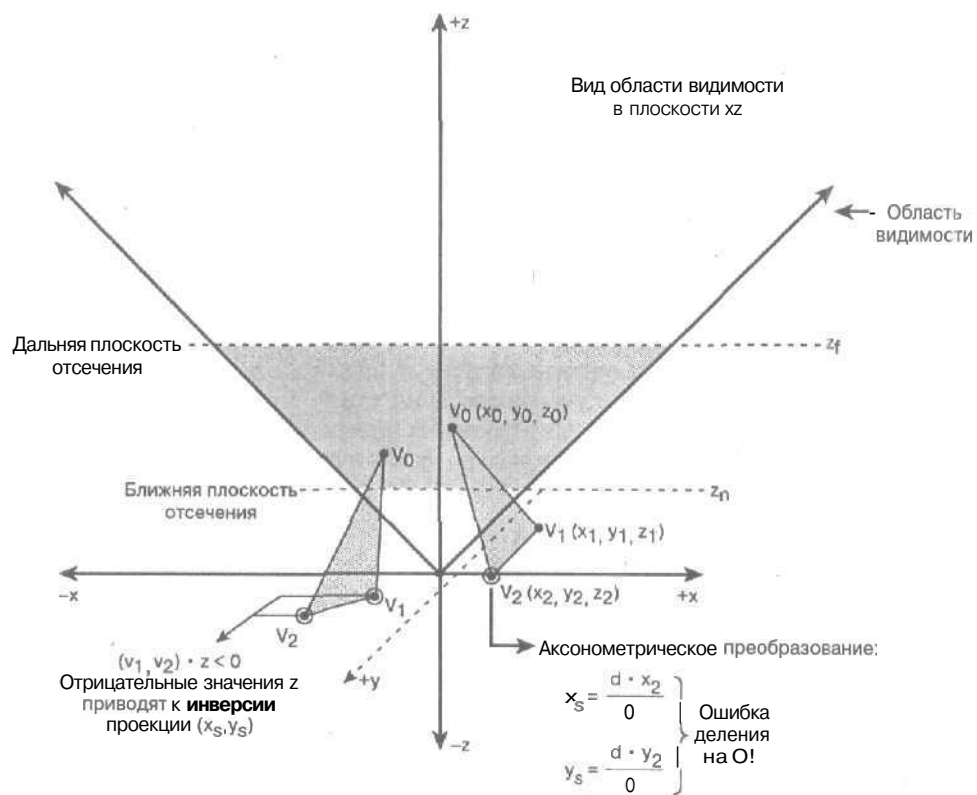


Рис. 10.1. Проецирование вершин с отрицательной или нулевой координатой недопустимо

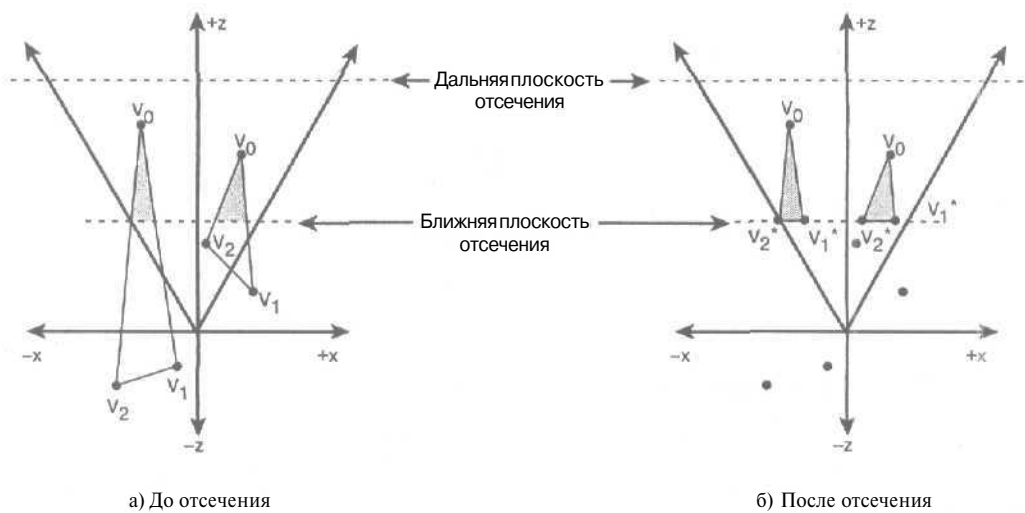


Рис. 10.2. Усечение длинных многоугольников

## Трехмерное отсечение в пространстве объекта

Если в игре имеются многоугольники произвольных размеров (рис. 10.2), то трехмерного отсечения никак не избежать. Можно, конечно, попытаться обойтись тем, что многоугольники, которые полностью находятся позади области ближнего отсечения, будут просто отбраковываться и игнорироваться, если они достаточно малы (рис. 10.3). Однако если игровой персонаж передвигается по определенному уровню, то этот прием может и не сработать.

Как минимум, необходимо выполнить усечение всех многоугольников по границам ближней плоскости отсечения. В результате усеченные многоугольники могут оказаться не треугольниками, а четырехугольниками. Поэтому их снова придется разбивать на треугольники, а затем заново помещать в список *визуализации*. Худший случай — когда по границам ближней плоскости придется усекать все многоугольники. Однако в ходе реализации этого процесса возникают дополнительные проблемы. О проблеме, связанной с разбиением на треугольники, уже упоминалось, но нужно также отсекать координаты текстуры, извлекать всю информацию, касающуюся затенения по *Гуро*, вычислять новые нормали в вершинах. Все это выглядит довольно драматично, однако поверьте — мы разберемся со всеми проблемами.

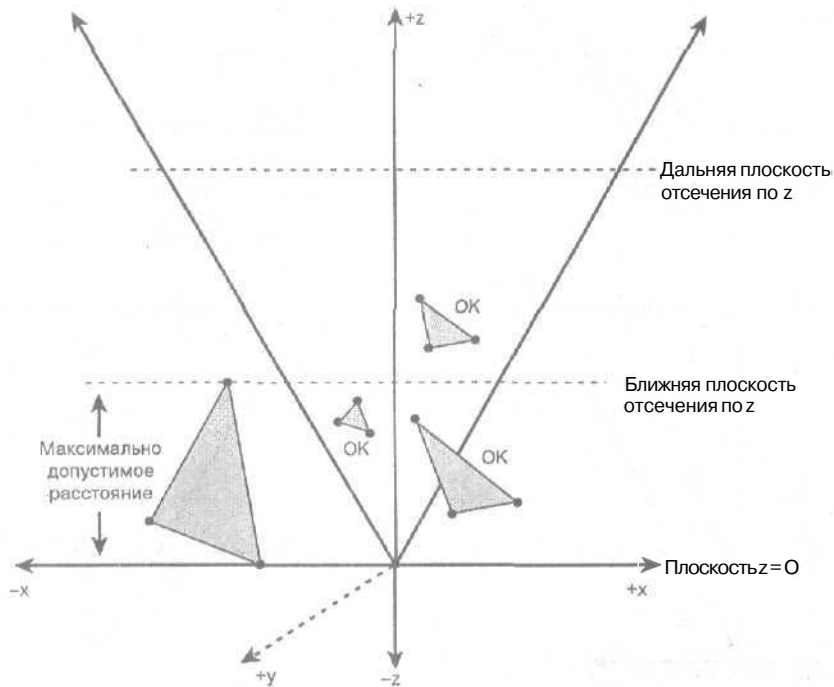


Рис. 10.3. Мимо небольших многоугольников можно незаметно проскользнуть

С другой стороны, нам предстоит узнать, что отсечение по остальным пяти *плоскостям*, ограничивающим область обзора, выполнять необязательно. Обратите внимание на рис. 10.4. На нем изображен многоугольник, отсеченный дальней плоскостью  $z$ . Есть ли какая-то польза от *этой* процедуры? Никакой, только время зря на нее потратим. Поэтому многоугольники, которые находятся за пределами дальней плоскости отсечения, будут попросту игнорироваться независимо от их размеров. Другими словами, если многоугольник полностью *выходит* за дальнюю плоскость отсечения, мы отбраковываем его, не пытаясь усечь.

Аналогичное утверждение можно сделать по поводу верхней, нижней, правой и левой плоскостей отсечения, **ограничивающих** область обзора. Отсечение по этим плоскостям — это напрасная трата времени. Причина в том, что эта операция требует некоторого времени, и здесь как раз уместно вспомнить, зачем оно нужно. **Отсечение** выполняется, чтобы полученные в его результате **проекции** координат не выходили за рамки двумерной области обзора. Однако это можно сделать и в пространстве изображений в процессе растеризации, что может оказаться таким же или еще более эффективным. Кроме того, нам не придется разбивать на треугольники многоугольники, усеченные по верхней, нижней, правой и левой плоскостям, ограничивающим область обзора,

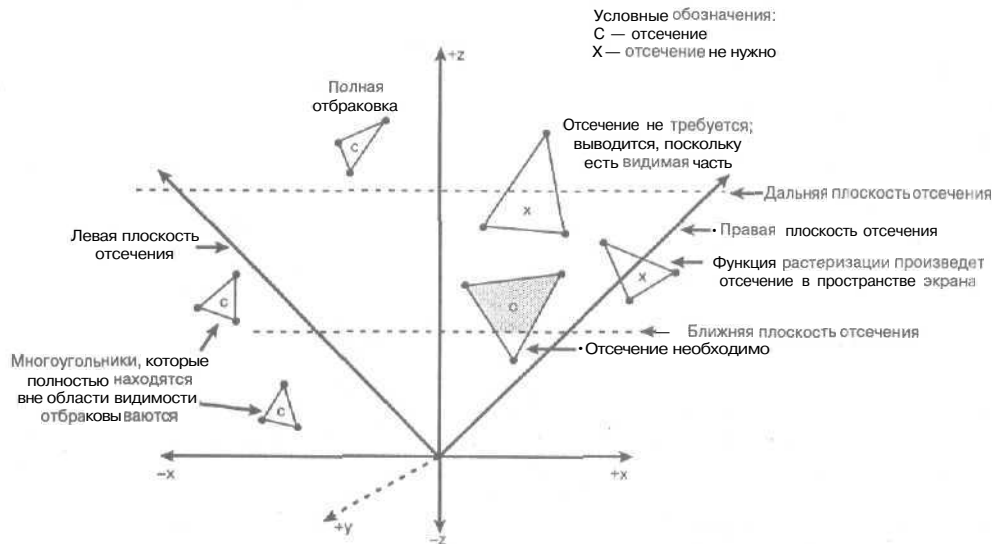


Рис. 10.4. Трехмерное **отсечение** следует выполнять по всем плоскостям, ограничивающим область обзора

Конечно же, если вы хотите реализовать такое отсечение (хотя бы просто для соблюдения аккуратности), я буду только рад и расскажу, как это сделать. Однако сейчас я собираюсь поработать над перечисленными ниже этапами отбраковки и отсечения (рис. 10.5).

1. Отбраковка всех объектов (по возможности; в игре может просто не быть объектов системного уровня или больших ландшафтов).
2. Удаление обратных поверхностей.
3. Отсечение всех многоугольников по границам области обзора. Полное отсечение и разбиение многоугольников выполняется только по ближней плоскости отсечения. По отношению к другим пяти плоскостям принимается **упрощение**, которое состоит в том, что многоугольник либо выводится полностью, либо отбраковывается. Это означает, что многоугольник, который полностью или частично находится в области обзора, обрабатывается. Если же многоугольник полностью выходит за область обзора, он отбраковывается.
4. Обработка всех неотбракованных многоугольников конвейером визуализации, при которой треугольники, выходящие за рамки двумерной области обзора и экрана, отсекаются на этапе растеризации.

Эту систему легко реализовать, она прекрасно работает и очень производительна.

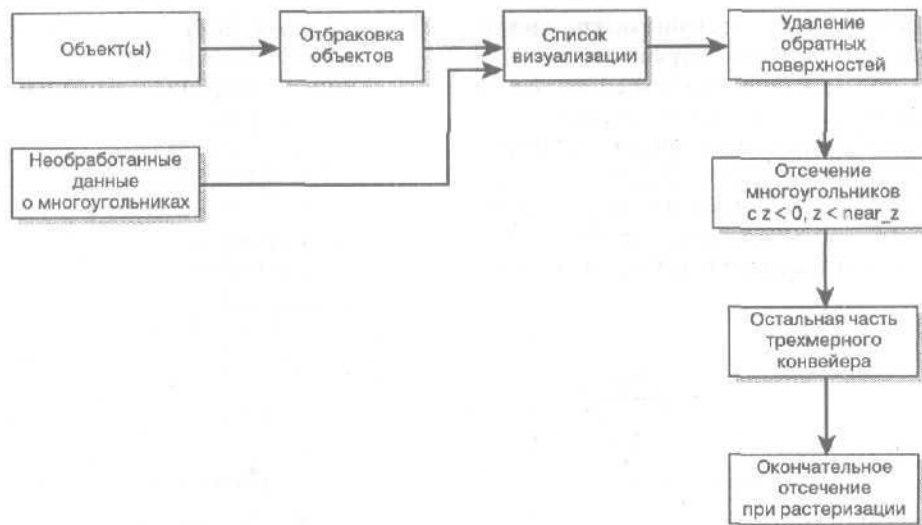


Рис. 10.5. Этапы конвейера трехмерного отсечения

## Отсечение в пространстве изображений

Давайте вспомним, что же именно подразумевается под отсечением в пространстве изображений. Ситуация, в которой применяется эта **операция**, проиллюстрирована на рис. 10.6. Многоугольник готов к растеризации, однако он частично выходит за пределы двумерной области, ограничивающей пространство экрана. Если выход происходит в вертикальном направлении, то это не проблема. В этом случае цикл растеризации просто начинается ниже верхней границы, там, где треугольник начинает перекрываться с экраном. **Соответствующий** код занимает не более одной строки. Если выход за рамки происходит по горизонтали, вступает в игру другой принцип растеризации, при котором производится отсечение каждой строки развертки, **являющейся базовым элементом**, участвующим в процессе визуализации. Однако выполнить указанные действия проще, чем в случае отсечения каждого треугольника всеми четырьмя плоскостями, образующими область обзора. При этом в конвейер могут быть добавлены новые многоугольники, возникает необходимость выполнить отсечение координат текстуры, повторное вычисление нормалей и т.д. Довольно неприятно также отсекал многоугольники по ближней плоскости.

С другой стороны, в пространстве изображений не хотелось бы производить никаких операций отсечения, в результате которых могла бы быть утрачена информация о трехмерном размещении объектов. Это весьма важный момент, особенно при работе с буфером глубины и т.п. Интерполируя вдоль сторон многоугольников величины, зависящие от координаты  $z$ , следует проявлять осторожность и, как минимум, сохранять информацию о  **$z$ -компонентах** вершин. Когда дело дойдет до растеризации, может оказаться, что *невозможно* выполнить отсечение в экранном пространстве на основе информации только о двух координатах. Рассмотрим, как все происходит, и позаботимся о том, чтобы в дальнейшем все работало корректно.

## Теоретические основы алгоритмов отсечения

Вскользь замечу, что именование алгоритмов — распространенная проблема в компьютерной графике. Кто-то **решает**, что он придумал новый "алгоритм", публикует его, и с

этого момента алгоритм получает имя (кстати, часто бывает, что это не входит в намерения автора. Он просто пишет статью, на которую потом ссылаются по имени ее автора).

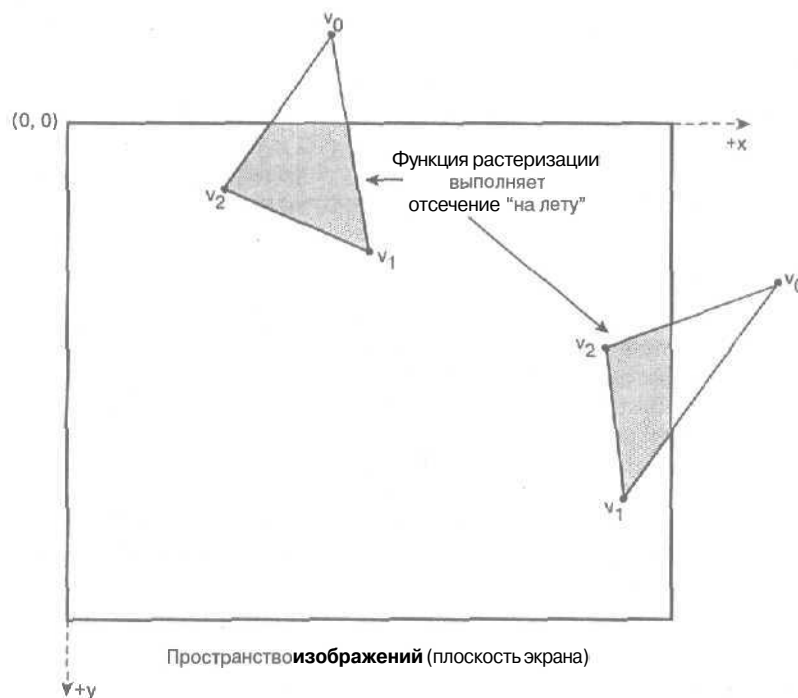


Рис. 10.6. Отсечение в пространстве изображений на этапе растеризации

Отсечение — это очень простая операция, и странно было бы даже подумать, что в этой области можно было бы что-нибудь "изобрести". Каждый алгоритм отсечения — это просто небольшая вариация поиска места пересечения линий, основанная на здравом смысле. Лично я стараюсь разработать такие алгоритмы, чтобы они соответствовали поставленным задачам. При этом иногда что-то приходится менять, а что-то — нет. Суть того, что я пытаюсь сделать, — это перечислить использующиеся алгоритмы отсечения по присвоенным им именам, однако, пожалуйста, не думайте, что описанный способ изложения — единственно возможный. Уверен, что многое из того, что я сейчас расскажу, вам уже известно. Иногда приходится проводить многие часы, пытаюсь постичь суть какой-то проблемы, а потом оказывается, что кто-то уже давно решил очень похожую. Что касается отсечения, то, честно говоря, хотелось бы, чтобы названия алгоритмов носили несколько более описательный характер, вот и все. Иногда даже трудно понять, то ли это названия алгоритмов, то ли юридических контор.

По моему мнению, для того, чтобы алгоритму присваивалось имя, он должен обладать определенным уровнем сложности и быть далеко неочевидным для специалиста среднего уровня, работающего в данной области. Полагаю, что в честь изобретателя следовало бы назвать такой алгоритм, как бинарное разбиение пространства (Binary Space Partitioning), поскольку он совершенно неочевиден. В то же время, если вы пытаетесь выполнить отсечение многоугольника, то через час-два вполне можно разобраться в том, как при этом будет преобразована каждая линия, как нужно классифицировать вершины, выполнять кодирование битов и производить параметрическое представление линий. Знаете что? Просто сами подумайте обо всем том, о чем я собираюсь вам рассказать!

Основная идея предыдущего абзаца заключается в том, что вам самим стоит испытать новые подходы. Девяносто девять процентов всего того, до чего вы *додумались*, было придумано до вас, однако не исключено, что что-то важное пока еще не было опубликовано. Поэтому просто пользуйтесь этими приемами и получайте удовольствие!

Я собираюсь описать некоторые из самых популярных алгоритмов отсечения с теоретической точки зрения, а позже мы реализуем их и приспособим для наших потребностей. Наш алгоритм будет представлять собой гибрид множества концепций, поскольку он составлен так, чтобы с его *помощью* можно было решать встающие перед нами задачи. Начнем с *рассмотрения* вопросов, лежащих в основе отсечения.

## Основы отсечения

Отсечение многоугольников сводится к отсечению прямых, которое, в свою очередь, сводится к определению того, находится ли данная точка прямой в некоторой области, независимо от того, задана ли она в двумерном или трехмерном пространстве. В общем виде проблема проиллюстрирована на рис. 10.7. Есть набор линий, входящих в состав многоугольников, которые на некотором этапе проецируются на плоскость. Необходимо *определить*, вписываются ли эти элементы в некоторую область. При этом проще определить, *лежат ли* в этой области отдельные точки. Другими словами, дело обстоит так. Все треугольники определяются тремя своими вершинами, которыми задаются их стороны. Конечно же, стороны тоже *важны*, поскольку с их помощью объект становится "заполненным" или "реальным", но на самом деле важны именно вершины. Таким образом, на самом нижнем уровне необходимо определить, находится ли точка  $(x, y)$  или  $(x, y, z)$  в пределах заданной области или вне нее. Для этого производится *тестирование точки* (point testing).

### Тестирование точки в двух измерениях

Допустим, в двумерном пространстве задана точка  $p_0(x, y)$  и прямоугольная область, ограниченная значениями  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$  и  $y_{max}$  (рис. 10.7). Точка находится внутри области, если выполняются неравенства  $(x_{min} \leq x < x_{max}) \wedge (y_{min} < y \leq y_{max})$ .

### Тестирование точки в трех измерениях

#### Частный случай 1: объем, ограниченный прямоугольным параллелепипедом

Пусть в трехмерном пространстве задана точка  $p_0(x, y, z)$  и область в форме прямоугольного параллелепипеда, ограниченная значениями  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$ ,  $y_{max}$ ,  $z_{min}$  и  $z_{max}$  (рис. 10.8). Точка находится внутри области, если выполняются неравенства  $(x_{min} \leq x \leq x_{max}) \wedge (y_{min} \leq y < y_{max}) \wedge (z_{min} < z \leq z_{max})$ .

Может возникнуть вопрос: зачем рассматривать отсечение по области, имеющей форму прямоугольного параллелепипеда, если известно, что область обзора имеет форму четырехгранной усеченной пирамиды? Это так, однако, следует помнить о том, что мы обсуждаем аксонометрическую проекцию, которая воздействует на область обзора, как бы "выпрямляя" ее и преобразуя в прямоугольный параллелепипед. В таком случае область отсечения становится очень простой, и для того, чтобы выполнить процедуру отсечения, достаточно произвести тривиальные операции сравнения. Кроме того, никто не утверждал, что отсечение обязательно нужно *выполнять* по границам области обзора. Возможно, понадобится произвести отсечение лазерных лучей, пронизывающих объект в форме куба или прямоугольного параллелепипеда, расположенный где-нибудь в игровом пространстве.

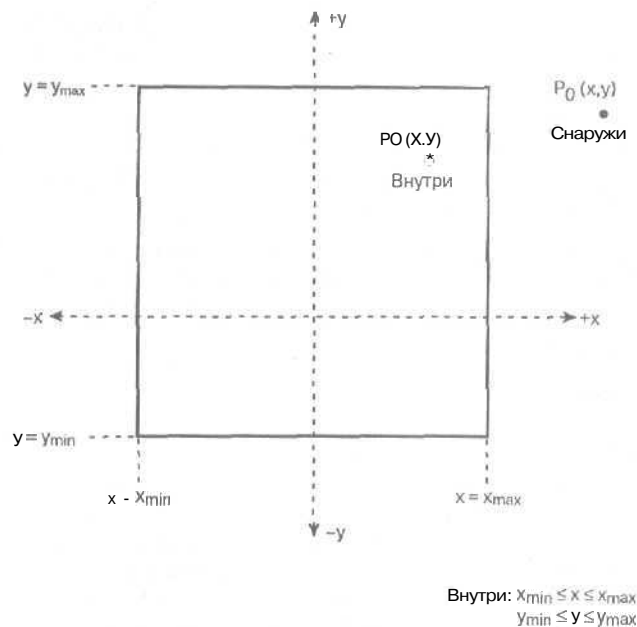


Рис. 10.7. Тестирование точки в двумерном пространстве

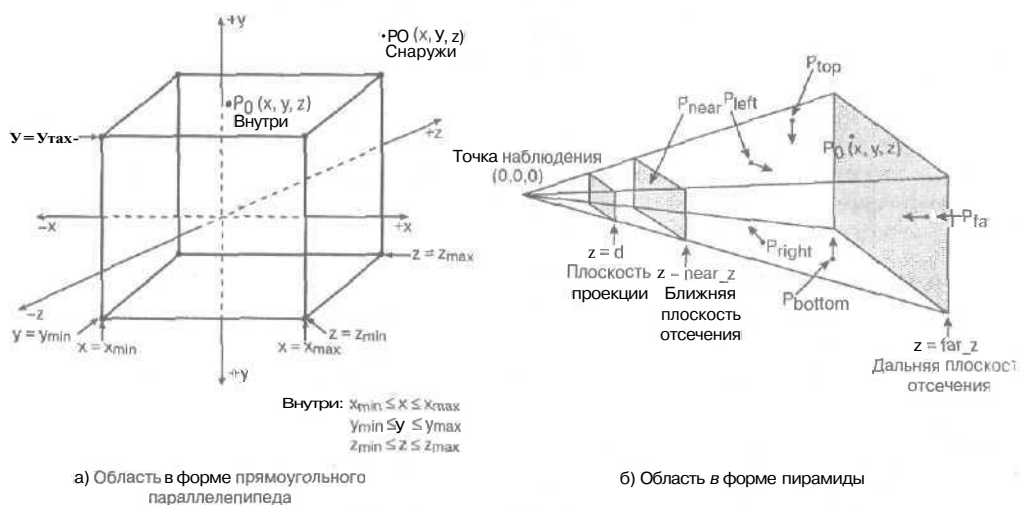


Рис. 10.8. Тестирование точки в трехмерном пространстве

### Частный случай 2: объем, ограниченный призмой

Это именно тот случай, который мы собираемся реализовать. Ситуация стандартная: есть шесть плоскостей, ограничивающих область обзора, которым мы присвоим имена  $P_{\text{top}}$ ,  $P_{\text{bottom}}$ ,  $P_{\text{right}}$ ,  $P_{\text{left}}$ ,  $P_{\text{far}}$  и  $P_{\text{near}}$ . Кроме того, предположим, что эти плоскости заданы таким образом, что их нормали направлены внутрь области обзора. Тогда тестирование точки будет заключаться в следующем. Для данной точки  $p_0(x, y, z)$  производится проверка того, в каком полупространстве она находится. Если эта точка расположена в полупро-

пространстве, куда направлена нормаль, она может попасть в область обзора. В противном случае точка  $p_0$  наверняка лежит за пределами области обзора. Напомним, что для того, чтобы проверить, *принадлежит* ли точка данной плоскости, или определить, в какой части пространства она находится, необходимо подставить координаты точки в уравнение плоскости (эта процедура описана в главах 4, "Запутанный мир математики", и 5, "Создание математической библиотеки"). Предположим, что у нас имеется оператор  $HS(p, pln)$ , который для заданной точки  $p$  и плоскости  $pln$  возвращает следующие значения: +1, если точка расположена в положительном полупространстве, -1 — если она находится в отрицательном полупространстве и 0, если точка лежит на плоскости.

Таким образом, задача сводится к проверке выполнения условия:

$$\begin{aligned} & (HS(p_0, P_{top}) > 0) \wedge (HS(p_0, P_{bottom}) > 0) \wedge \\ & (HS(p_0, P_{right}) > 0) \wedge (HS(p_0, P_{left}) > 0) \wedge \\ & (HS(p_0, P_{near}) > 0) \wedge (HS(p_0, P_{far}) > 0). \end{aligned}$$

## Основы отсечения прямых

Теперь мы можем определить, находятся ли точки в заданной (двух- или трехмерной) области отсечения. Следующий этап состоит в том, чтобы подняться еще на один уровень и научиться выполнять отсечение прямых или отрезков, соединяющих вершины треугольников. Таким образом, нам понадобится инструментарий, позволяющий находить точки пересечения отрезков с прямыми и (или) другими отрезками. Этот вопрос тоже рассматривался в главах 4 и 5, поэтому мы не станем заниматься им столь же подробно.

Отсечение многоугольников позволяет нам подняться на еще один уровень. Чтобы ничего не испортить, нужно подумать над тем, как производить отсечение прямой, которая пересекается с другой прямой или с плоскостью. Весь алгоритм отсечения представляет собой не что иное, как итерацию или рекурсивное применение процедуры отсечения отрезков по границам некоторой области, которая последовательно повторяется для все новых сторон многоугольников. Конечно же, при разработке этого алгоритма нужно постараться свести всю работу к минимуму, однако в его основе в любом случае будет отсечение — двумерное, трехмерное, любое.

## Отсечение отрезков на плоскости

Вернемся к проблеме отсечения отрезков. Сначала рассмотрим двумерный случай. Обратите внимание на рис. 10.9, на котором изображен отрезок, соединяющий точки  $p_0(x_0, y_0)$  и  $p_1(x_1, y_1)$ . Нужно выполнить отсечение этого отрезка по вертикали  $X = x_L$ .

### НА ЗАМЕТКУ

В большинстве случаев нам не придется выполнять отсечение сторон треугольника произвольными прямыми, поэтому в принципе достаточно рассмотреть отсечение вертикальной и горизонтальной границами.

Для решения поставленной задачи достаточно записать уравнение прямой, которой принадлежит отсекаемый отрезок, в параметрическом виде, а затем подставить в него значение  $x_L$ .

Для произвольной точки  $p$ , принадлежащей отрезку, можно записать векторное уравнение  $p = p_0 + (p_1 - p_0) \cdot t$ . Расписывая это уравнение для каждой из компонент, получим:

$$x = x_0 + (x_1 - x_0) \cdot t, \quad 0 \leq t \leq 1,$$

$$y = y_0 + (y_1 - y_0) \cdot t, \quad 0 \leq t \leq 1.$$

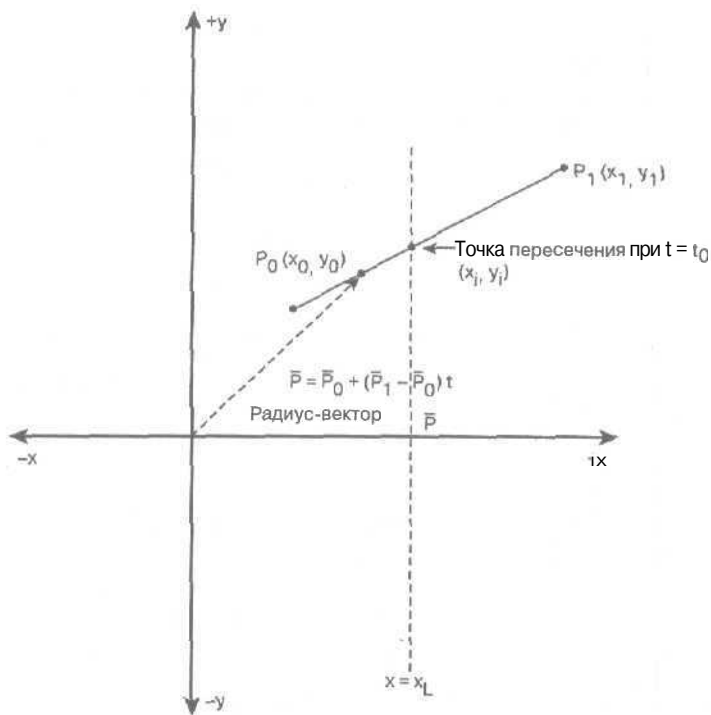


Рис. 10.9. Простое отсечение отрезка

Чтобы найти точку пересечения этого отрезка с вертикальной прямой, подставляем в левую часть приведенного выше уравнения значение  $x$ , которым определяется эта вертикальная прямая (в нашем случае это  $x = x_L$ ), и решить его относительно параметра  $t$ . Если полученный результат лежит в интервале  $t \in [0, 1]$ , то вертикальная прямая пересекает отрезок. В противном случае она его не пересекает. Вот как выглядит описанная выше процедура. Берем уравнение

$$x = x_0 + (x_1 - x_0) \cdot t, \quad 0 \leq t \leq 1$$

и подставляем в его левую часть значение  $x_L$ :

$$x_L = x_0 + (x_1 - x_0) \cdot t.$$

Затем решаем полученное уравнение относительно  $t$ :

$$t = \frac{(x_L - x_0)}{(x_1 - x_0)}.$$

Если полученное значение  $t$  находится между нулем и единицей, точка пересечения находится на отрезке. Напомним, что приведенное выше параметрическое уравнение справедливо для точек отрезка, лежащих между точками  $p_0$  и  $p_1$  (т.е. для  $t \in [0, 1]$ ). Другим значениям параметра  $t$  тоже соответствуют точки, которые лежат на прямой, соединяющей  $p_0$  и  $p_1$ , но эти точки уже не принадлежат отрезку. Предположим, что полученное значение  $t$  отвечает сформулированному условию. Далее следует подставить уравнение для координаты  $y$

$$y = y_0 + (y_1 - y_0) \cdot t$$

и получить координаты пересечения отрезка с вертикальной прямой  $x = x_L$ . Случай пересечения с горизонтальной прямой столь же прост, но вместо того, чтобы решать относительно параметра  $t$  уравнение для координаты  $x$ , нужно использовать уравнение для координаты  $y$ . Найдем точку пересечения отрезка с прямой  $y = y_L$ .

Подставляя значение  $y_L$  в левую часть уравнения для координаты  $y$ , получим

$$y_L = y_0 + (y_1 - y_0) \cdot t.$$

Решим это уравнение относительно  $t$ :

$$t = \frac{(y_L - y_0)}{(y_1 - y_0)}.$$

Как и в предыдущем случае, проверяем полученное значение  $t$ . Если выполняется условие ( $t \in [0, 1]$ ), то точка пересечения принадлежит отрезку и мы подставляем  $t$  в уравнение для  $x$ :

$$x = x_0 + (x_1 - x_0) \cdot t.$$

В результате мы находим координаты точки пересечения отрезка с прямой, которая на этот раз является горизонтальной.

Конечно же, здесь нужно руководствоваться здравым смыслом и выполнить определенные предварительные проверки. Например, если точки  $p_0$  и  $p_1$  совпадают, то нет необходимости выполнять проверку на пересечение. Аналогично, бессмысленно находить точку пересечения горизонтального отрезка с горизонтальной прямой.

### Отсечение отрезков в пространстве

Отсечение отрезков, заданных в трехмерном пространстве, ничуть не сложнее отсечения на плоскости. Все дело в том, как представить плоскости отсечения. Здесь не понадобятся никакие трюки, поэтому рассмотрим общий случай, когда в трехмерном пространстве задано уравнение плоскости отсечения и параметрическое уравнение прямой. Обратимся к рис. 10.10, который иллюстрирует нашу задачу. Сначала рассмотрим уравнение плоскости, которая определяется принадлежащей ей точкой и вектором нормали.

Пусть плоскость проходит через точку  $p_0(x_0, y_0, z_0)$  и для нее задан вектор нормали  $n = \{n_x, n_y, n_z\}$ . Тогда ее уравнение можно записать в виде

$$n_x(x - x_0) + n_y(y - y_0) + n_z(z - z_0) = 0.$$

Теперь рассмотрим параметрическое уравнение прямой, соединяющей точки  $p_1$  и  $p_2$ :

$$p = p_1 + (p_2 - p_1) \cdot t, \quad 0 \leq t \leq 1.$$

Это уравнение можно расписать покомпонентно:

$$x = x_1 + (x_2 - x_1) \cdot t,$$

$$y = y_1 + (y_2 - y_1) \cdot t,$$

$$z = z_1 + (z_2 - z_1) \cdot t.$$

Теперь нужно найти точку пересечения отрезка и плоскости. Этот вопрос рассматривался в главе 4, "Запутанный мир математики", но в конечном счете все сводится к тому, что выражения для координат  $x$ ,  $y$  и  $z$ , заданные в уравнении прямой, подставляются в уравнение плоскости. В результате мы получаем уравнение:

$$n_x((x_1 + (x_2 - x_1)t) - x_0) + n_y((y_1 + (y_2 - y_1)t) - y_0) + n_z((z_1 + (z_2 - z_1)t) - z_0) = 0.$$

Далее решаем это уравнение относительно параметра  $t$  и подставляем полученный результат в выражения для координат  $x$ ,  $y$  и  $z$ , чтобы найти сами координаты. Это мы

уже делали, поэтому опустим выкладки, а заодно и сэкономим немного места — надеюсь, идея понятна. Вот и все, что можно сказать по поводу отсечения. Как только мы научимся находить точку пересечения отрезка с прямой и плоскостью, можно считать, что проблема решена. Все, что нужно на самом деле, — это приспособить алгоритм к конкретной ситуации. Рассмотрим еще несколько дополнительных алгоритмов отсечения.

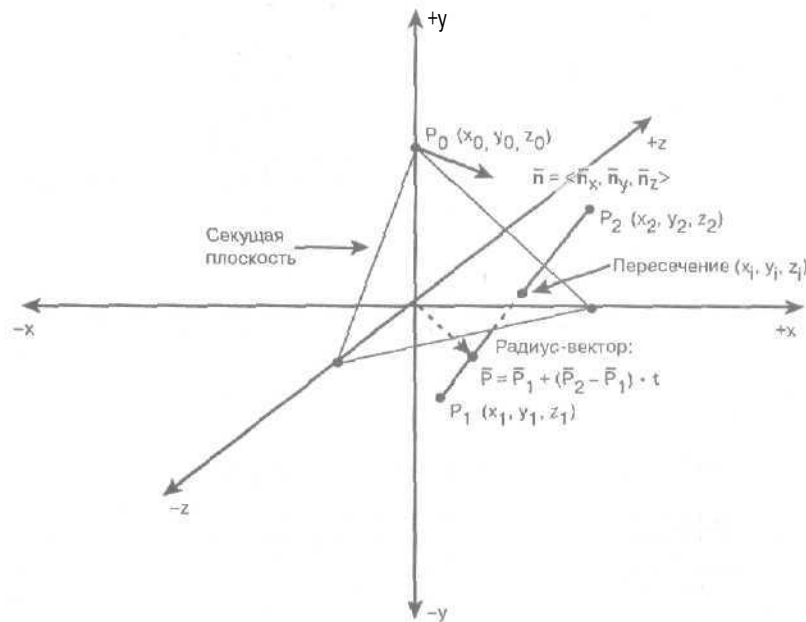


Рис. 10. 10. Отсечение отрезка плоскостью в трехмерном пространстве

## Отсечение по Кохену-Сазерленду

Алгоритм Кохена-Сазерленда (Cohen-Sutherland) является одним из самых популярных методов отсечения отрезков по границам прямоугольной (двумерной) области или (трехмерной) области, имеющей форму прямоугольного параллелепипеда. Алгоритм состоит из двух этапов.

### Этап классификации конечных точек

На первом этапе конечные точки всех отрезков, заданных в двумерном или трехмерном пространстве, классифицируются в зависимости от того, какое положение они занимают относительно границ. Для классификации используются битовые коды, в которых задаются значения FALSE или TRUE. Схема классификации для двумерного случая представлена на рис. 10.11.

Классифицируются обе точки  $p_0(x_0, y_0, z_0)$  и  $p_1(x_1, y_1, z_1)$  каждого отрезка. Для каждой из этих точек задается переменная с 4-битовым (а в трехмерном случае 6-битовым) кодом. Пусть вершине  $p_0$  соответствует переменная `bitcode0`, а переменной  $p_1$  — переменная `bitcode1`. Биты можно кодировать произвольными значениями, однако чаще всего для двумерной прямоугольной области они кодируются значениями, перечисленными в табл. 10.1.

Случай 1: отрезок игнорируется  
 Случай 2: необходимо отсечение  
 Случай 3: отрезок выводится  
 Случай 4: необходимо отсечение

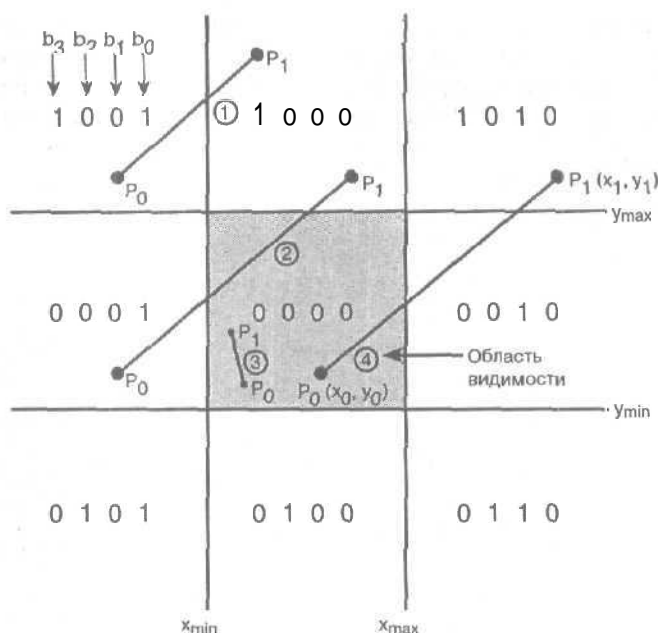


Рис. 10.11. Схема классификации с помощью битового кода при отсечении по Кохену-Сазерленду

Таблица 10.1. Кодирование битов в алгоритме Кохена-Сазерленда

Бит	Условие, которому соответствует значение TRUE
b3	$y > y_{\max}$ (точка расположена над верхней границей)
b2	$y < y_{\min}$ (точка расположена под нижней границей)
b1	$x > x_{\max}$ (точка расположена правее от правой границы)
b0	$x < x_{\min}$ (точка расположена левее от левой границы)

После того как проклассифицированы конечные точки всех отрезков, начинается второй этап алгоритма, когда производится анализ и выполняется отсечение,

### Этап обработки

Этап обработки начинается с определения того, какие отрезки полностью находятся внутри или вне области обзора. Приятная особенность этого алгоритма состоит в том, что в ходе его выполнения производятся предварительные вычисления, позволяющие сэкономить время и избежать применения многих условных операторов. Поскольку в нашем распоряжении для каждой вершины есть слово, состоящее из кодовой последовательности битов, появляется возможность применения побитовых операций, позволяющих получить ответ сразу на несколько вопросов. Предположим, что нужно узнать, находится ли отрезок за пределами области отсечения. Для этого следует проверить, находятся ли обе вершины в верхней, нижней, правой или левой полуплоскости, и здесь не обойтись без большого количества условных операторов! Однако с помощью битовых кодов все делается тривиально, поскольку можно сформировать логические выражения и параллельно задавать несколько вопросов. Два самых простых случая — когда отрезок полностью отбрасывается или полностью попадает в область обзора. Рассмотрим их подробнее.

Тест 1. Если выполняются условия ( $\text{bitcode0}=0$ ) и ( $\text{bitcode1}=0$ ), то отрезок, соединяющий точки  $P_0$  и  $P_1$ , полностью принадлежит области обзора и отсечение не требуется (рис. 10.11).

Далее применяем побитовый оператор И к переменным  $\text{bitcode0}$  и  $\text{bitcode1}$  и проверяем полученный результат:

$$\text{bc} = (\text{bitcode0} \& \text{bitcode1})$$

Тест 2. Если ( $\text{bc}>0$ ), то обе конечные точки отрезка находятся за пределами, по крайней мере, одной границы области обзора, и его можно попросту игнорировать (см. рис. 10.11).

Если тесты 1 и 2 не дали положительных результатов, начинается проблемная часть. Может оказаться так, что отрезок пересекает границы области обзора в одной или двух точках (см. рис. 10.11). Есть несколько подходов к обработке этой ситуации. Один из методов заключается в том, чтобы выполнить отсечение отрезка по обеим границам, которые он пересекает, в результате чего отрезок разбивается на две или три части. Далее этот алгоритм применяется к новым отрезкам. Для нахождения точек пересечения можно применить алгоритм деления или воспользоваться решением "в лоб" (что далеко не всегда плохо).

## Отсечение по Сайрусу-Беку и Лянгу-Барскому

Алгоритм, который предлагается вашему вниманию, несколько более общий по сравнению с предыдущим. Он работает с параметрическими уравнениями прямых и обобщенными выпуклыми (двух- или трехмерными) областями отсечения. Этот алгоритм работает с параметрическим представлением прямых (как наиболее естественным), и для поиска точек пересечения используются не их явные координаты  $(x, y)$  или  $(x, y, z)$ , а значения параметра  $t$ . Затем на основе полученного значения параметра можно определить все интересующие нас параметры более высокого уровня. Это избавляет данный алгоритм от ряда низкоуровневых деталей, присущих алгоритму Кокена-Сазерленда.

Рассмотрим рис. 10.12, который иллюстрирует двумерный случай. Пусть вертикальные границы задаются значениями  $x_{\min}$  и  $x_{\max}$ , горизонтальные —  $y_{\min}$  и  $y_{\max}$ , а отрезок принадлежит прямой, параметрическое уравнение которой имеет вид

$$P = P_1 + (P_2 - P_1) \cdot t, \quad 0 \leq t \leq 1.$$

Распишем это векторное уравнение по компонентам, обозначив  $dx = (x_2 - x_1)$  и  $dy = (y_2 - y_1)$ . В результате мы получим систему уравнений

$$x = x_1 + dx \cdot t,$$

$$y = y_1 + dy \cdot t.$$

Чтобы отрезок полностью находился в области отсечения, должны соблюдаться следующие неравенства:

$$x_{\min} \leq x_1 + dx \cdot t \leq x_{\max},$$

$$y_{\min} \leq y_1 + dy \cdot t \leq y_{\max},$$

что после упрощений дает:

$$x_{\min} \leq x_1 + dx \cdot t,$$

$$-(x_1 + dx \cdot t) \geq -x_{\max},$$

$$-(y_1 + dy \cdot t) \geq -y_{\max},$$

$$y_{\min} \leq y_1 + dy \cdot t.$$

Выполним еще пару преобразований, чтобы члены  $dx \cdot t$  и  $dy \cdot t$  оказались с одной стороны неравенств, после чего, умножая неравенства на  $-1$  со сменой знака неравенства на противоположный, получим **следующие** неравенства отсечения.

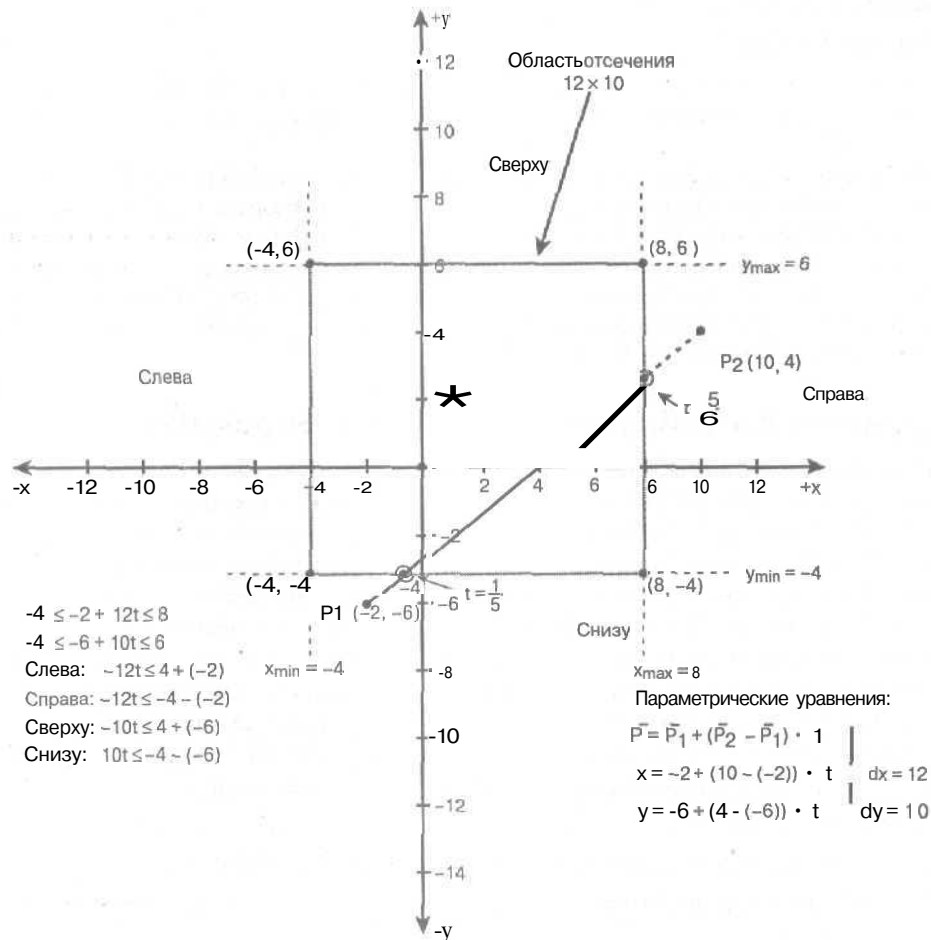


Рис. 10.12. Схема отсечения по Сайрусу-Беку (Cyrus-Beck) и Лянгу-Барскому (Liang-Barsky)

#### Уравнение 10.1. Неравенства отсечения

Неравенство для левой стороны	$n = 0:$	$-dx \cdot t \leq (-x_{\min} + x_1)$
Неравенство для правой стороны	$n = 1:$	$dx \cdot t \leq x_{\max} - x_1$
Неравенство для верхней стороны	$n = 2:$	$dy \cdot t \leq y_{\max} - y_1$
Неравенство для нижней стороны	$n = 3:$	$-dy \cdot t \leq -y_{\min} + y_1$

#### СОВЕТ

Я потерял очень много времени, добиваясь того, чтобы эти неравенства были правильными. Меняя направление знака **неравенства**, легко допустить ошибку, поэтому все преобразования нужно выполнять постепенно. Запомните: при умножении или делении обеих частей **неравенства** на отрицательное число знак неравенства меняется на противоположный.

Как видите, все неравенства имеют вид  $p_n \cdot t < q_n$ , где  $n$  — номер соотношения. Конечно же, предварительно следует убедиться, что величины  $dx$  и  $dy$  отличны от нуля. Если же  $dx = 0$  или  $dy = 0$ , то отрезок, соответственно, параллелен вертикальным или горизонтальным границам и не может их пересекать. Далее, обозначим через  $t_n$  значение параметра  $t$ , найденное из приведенных выше соотношений, если знак неравенства заменить в них знаком равенства. Заметим, что точка пересечения принадлежит отрезку при условии  $0 < t_n < 1$ . Если же  $t_n < 0$  или  $t_n > 1$ , то точка находится на прямой, на которой лежит отрезок, но самому отрезку эта точка не принадлежит. Поэтому, если данное условие выполняется для всех  $n$ , отрезок полностью находится за рамками области отсечения. Независимо от используемого алгоритма, можно выделить четыре перечисленных ниже частных случая взаимного расположения отрезка и выпуклой области отсечения.

1. Отрезок лежит за пределами области отсечения. Это происходит, когда  

$$(t_0 < 0 \ \&\& \ t_1 < 0) \ || \ (t_0 > 1 \ \&\& \ t_1 > 1) \ ||$$

$$(t_2 < 0 \ \&\& \ t_3 < 0) \ || \ (t_2 > 1 \ \&\& \ t_3 > 1)$$
2. Отрезок полностью принадлежит области отсечения. Это происходит при условии:  

$$((t_0 \leq 0 \ \&\& \ t_1 \geq a) \ || \ (t_0 \geq 1 \ \&\& \ t_1 \leq 0)) \ \&\&$$

$$((t_2 \leq 0 \ \&\& \ t_3 \geq 1) \ || \ (t_2 \geq 1 \ \&\& \ t_3 \leq 0))$$
3. Отрезок пересекает границу области отсечения в одной точке. Это происходит при условии, что одно из значений  $t_n$  принадлежит интервалу  $0..1$ , а все остальные — лежат за его пределами.
4. Отрезок пересекает границы области отсечения в двух точках. Этот случай самый сложный и требует более подробного рассмотрения.

Введем величины

$$t_{\max} = \max_{p_i < 0} (0, t_i)$$

и

$$t_{\min} = \min_{p_i > 0} (1, t_i).$$

Если  $t_{\max} > t_{\min}$ , то отрезок полностью выходит за рамки области отсечения. В противном случае он отсекается в двух местах, соответствующих тем значениям параметра  $t_n$ , которые находятся в числовом интервале  $0..1$ . Чтобы найти точки пересечения, нужно эти значения подставить в параметрическое уравнение отрезка:

$$pi_1 = p_1 + (p_2 - p_1) \cdot t_{\max},$$

$$pi_2 = p_1 + (p_2 - p_1) \cdot t_{\min}.$$

Я согласен, что этот алгоритм кажется кошмаром и трудно воспринимается человеком, однако его достоинство в том, что он хорошо подходит для компьютера. Он замечательно работает, причем намного лучше алгоритма Кохена-Сазерленда, потому что основан на параметрических уравнениях прямых и легко обобщается для трехмерного случая.

### Пример использования алгоритма

Закрепим понимание алгоритма. Для этого рассмотрим простой пример (см. рис. 10.12), чтобы убедиться в работоспособности алгоритма. Итак, у нас имеются следующие исходные данные:

$$x_{\min} = -4, x_{\max} = 8$$

$$y_{\min} = -4, y_{\max} = 6$$

$$p1=(-2,-6), p2=(10,4)$$

$$p = p1 + (p2-p1)*t$$

$$x = -2 + (10 - (-2))*t = -2 + 12*t$$

$$y = -6 + (4 - (-6))*t = -6 + 10*t$$

Таким образом,  $dx = 12$  и  $dy = 10$ . Теперь приступим к выполнению алгоритма и найдем неравенства для каждой **границы** области отсечения:

$$\text{Слева: } -dx*t \leq (-x_{\min} + x1)$$

$$\text{Справа: } dx*t \leq (x_{\max} - x1)$$

$$\text{Сверху: } dy*t \leq (y_{\max} - y1)$$

$$\text{Снизу: } -dy*t \leq (-y_{\min} + y1)$$

Подставляя в эти соотношения конкретные значения, получим:

$$\text{Слева: } -12*t \leq (-(-4) + (-2)) - (-2)$$

$$\text{Справа: } 12*t \leq (8 - (-2)) - (-10)$$

$$\text{Сверху: } 10*t \leq (6 - (-6)) - (-12)$$

$$\text{Снизу: } -10*t \leq (-(-4) + (-6)) - (-2)$$

Определим значения  $p_n$  и  $q_n$  и вычислим все  $t_n$ :

$$\text{Слева: } -12*t \leq (-2), p0=-12, q0=2, t0=(q0/p0)=-1/6$$

$$\text{Справа: } 12*t \leq (10), p1=12, q1=10, t1=(q1/p1)=5/6$$

$$\text{Сверху: } 10*t \leq (12), p2=10, q2=12, t2=(q2/p2)=6/5$$

$$\text{Снизу: } -10*t \leq (-2), p3=-10, q3=-2, t3=(q3/p3)=1/5$$

Теперь для  $p_n < 0$  вычисляем значение

$$t_{\max} = \max(0, t_n) = \max(0, -1/6, 1/5) = 1/5$$

Для  $p_n > 0$  вычисляем значение

$$t_{\min} = \min(1, t_n) = \min(1, 5/6, 6/5) = 5/6$$

Далее проверяем, выполняется ли неравенство  $t_{\max} > t_{\min}$ . В этом случае отрезок находится за пределами области отсечения. Очевидно, что в рассматриваемом примере это не так. Теперь подставим значения параметров  $t_{\min}$  и  $t_{\max}$  в параметрическое уравнение прямой и в результате получим новый отрезок. Итак, подставляя значение  $t_{\max}$ , получим

$$x1' = -2 + 12*(1/5) = -2 + 2.4 = 0.4$$

$$y1' = -6 + 10*(1/5) = -6 + 2 = -4.0$$

а для значения  $t_{\min}$  находим

$$x2' = -2 + 12*(5/6) = -2 + 10 = 8$$

$$y2' = -6 + 10*(5/6) = -6 + 8.3 = 2.3$$

Таким образом, отсеченный фрагмент отрезка находится между точками  $p'_1(0.4, -4.0)$  и  $p'_2(8.0, 2.3)$ . Ирис. 10.12 подтверждает это.

Знаете, что забавно? Структура кода, присущая многим функциям отсечения, разработанным программистами, которые не знали этого алгоритма, все равно напоминает о нем. Это свидетельствует о том, что алгоритм действительно хорош!

Мы не станем заниматься формальной реализацией этого алгоритма. Однако позже, когда наступит черед оптимизации, возможно, мы еще вернемся к алгоритму Кохена-Сазерленда,

использовав его **для** двумерного отсечения. Может возникнуть вопрос: а применим ли этот алгоритм в трехмерном пространстве? Конечно, да. Чтобы обобщить его для случая с тремя измерениями, следует воспользоваться точно такими же процедурами. Отличие в **том**, что теперь неравенства указывают, в какой части полупространства находятся точки по отношению к плоскостям отсечения  $\sim$  в положительной или отрицательной, и для тестирования надо будет использовать, например, скалярное произведение векторов.

## Отсечение по Вейлеру-Азертону

Для понимания последнего из описываемых алгоритмов удобнее всего исходить из представления, что все многоугольники состоят из вершин и что каждая вершина либо принадлежит области отсечения, либо нет. Алгоритм составлен таким образом, что позволяет отсечь произвольный многоугольник по границам **любого** другого многоугольника (области отсечения). Согласно **общей** схеме, если сторона многоугольника пересекает область отсечения, то по отношению к этой области она является либо входящей, либо выходящей. На рис. 10.13 многоугольник, от которого отсекаются отдельные части, называется *предметным многоугольником* (subject polygon), а многоугольник, выступающий в роли области отсечения, — *многоугольником отсечения* (clipping polygon).

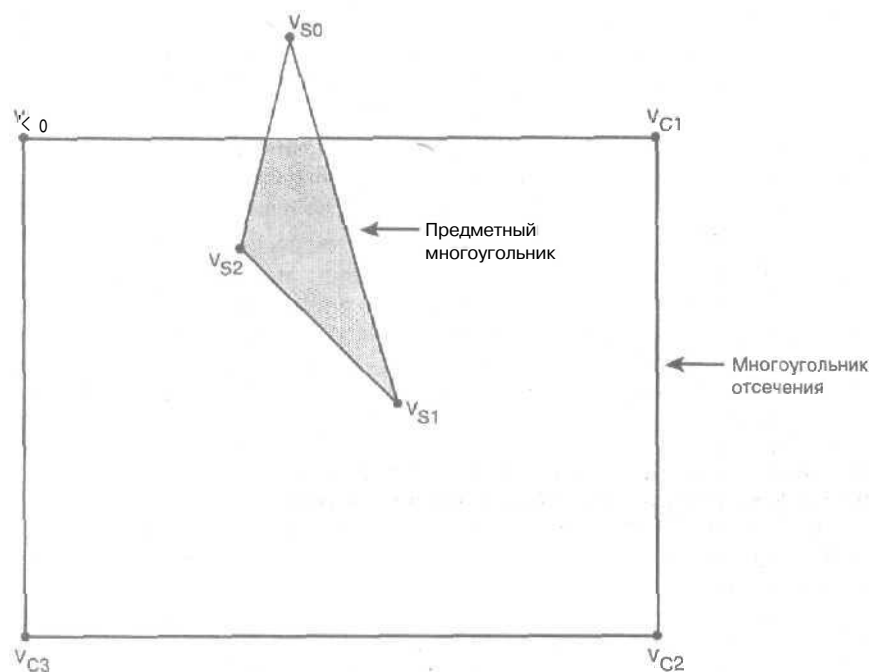


Рис. 10.13. Схема алгоритма Вейлера-Азертон (Weiler-Atherton)

Чтобы алгоритм работал правильно, многоугольники должны быть заданы с помощью определенным образом упорядоченного списка вершин, хранящегося в массиве или связанном списке. Предположим, что в обоих многоугольниках вершины упорядочены по ходу часовой стрелки. Работа алгоритма начинается с обработки отрезка, соединяющего вершины 0 и 1. Для этого отрезка **проверяется**, пересекает ли он каждую из сторон многоугольника отсечения. Если отрезки пересекаются, точка пересечения добавляется в

оба списка вершин — и тот, который отвечает предметному **многоугольнику**, и тот, который отвечает многоугольнику отсечения. Новая вершина вставляется в список между вершинами, **выступающими** в роли концов пересекаемого отрезка. При этом ее нужно пометить флагом входа или выхода, указывающим, входит ли сторона в область отсечения или выходит из нее. После того как списки вершин созданы, над получившимся в результате многоугольником выполняется такая процедура.

1. Находим в списке вершин предметного многоугольника первую по порядку входную точку пересечения. Далее создаем аналогичный список для **резльтирующего** (получившегося в результате отсечения) многоугольника и заносим туда эту точку в качестве первой вершины.
2. Продолжаем обработку предметного многоугольника до тех пор, пока не обнаружим следующую точку пересечения. Попутно добавляем каждую обработанную вершину в список вершин результирующего многоугольника. Следующая точка пересечения, которая нам встретится, по определению должна быть выходящей.
3. Исходя из расположения этой точки пересечения в списке вершин предметного многоугольника, производим обработку списка вершин многоугольника отсечения, отыскивая в нем *ту же* точку пересечения (чтобы синхронизировать списки).
4. Далее продолжаем обход списка вершин многоугольника отсечения, и заносим каждую вершину этого списка в список вершин результирующего многоугольника. Продолжаем эту процедуру до тех пор, пока не дойдем до **следующей** точки пересечения. Если все выполнено правильно, то это должна быть точка входа.
5. Если точка пересечения, о которой шла речь на шаге 4, не совпадает с первой вершиной результирующего **многоугольника**, возвращаемся к этапу 2 (мы стремимся к тому, чтобы замкнуть результирующий многоугольник).
6. Если точка пересечения, о которой шла речь на шаге 4, совпадает с первой вершиной **резльтирующего** многоугольника, получаем замкнутый многоугольник (правда, это **еще** не означает, что алгоритм завершен). Если все точки входа предметного многоугольника включены в список вершин результирующего многоугольника (чтобы это **определить**, следует использовать счетчик или флаги), работу алгоритма можно считать завершенной. В противном случае мы переходим к следующей **входной** точке предметного многоугольника и **возвращаемся** к шагу 2.

Описанный алгоритм замечательно работает и позволяет выполнить отсечение любого многоугольника по границам любого другого многоугольника. На рис. 10.14 схема описанного алгоритма проиллюстрирована на простом примере, когда алгоритм состоит из небольшого количества шагов.

Лично мне не по душе все эти подготовительные этапы и многочисленные переходы в списке вершин. Кроме того, напомним, что мы имеем дело с треугольниками, да и **вообще** не можем себе позволить применять длинные алгоритмы. В большинстве случаев придется модифицировать этот алгоритм с помощью идей, **позволяющих** многократно ускорить его работу. Опишем другой алгоритм, в основе которого лежит идея о точках входа и выхода, как и в алгоритме Вейлера-Азертонна.

Первым делом рассмотрим случай отсечения одного многоугольника (треугольника) по одной стороне или грани области отсечения (впоследствии можно обобщить алгоритм на случай нескольких **границ** двумерной или трехмерной области, повторяя его для каждой из границ). Для начала проклассифицируем вершины в зависимости от того, где они находятся — **внутри** области отсечения или вне нее. Ес-

ли вершина находится в области отсечения, она заносится в список вершин результирующего многоугольника; в противном случае не выполняется никаких действий, и происходит переход к следующей вершине. Если начало и окончание отрезка, принадлежащие одной грани, находятся в разных состояниях (т.е. одна из них принадлежит области отсечения, а другая — нет), находим точку пересечения и добавляем ее в результирующий список. По сути, мы просто одну за другой перебираем стороны многоугольника. Если получилось так, что алгоритм начался с обработки внутренней точки, то это замечательно. При каждом выходе за пределы области отсечения образуется точка пересечения, и алгоритм продолжает работу. Далее, когда мы дойдем до той стороны многоугольника, которая снова входит в область отсечения, образуется еще одна точка пересечения, и снова продолжается выполнение алгоритма. На самом деле, это упрощенная версия алгоритма Вейлера-Азертон, однако такая, которая может работать в реальном времени, "на лету". Именно этим методом в том или ином виде я пользуюсь в большинстве случаев, поскольку излишне создавать сложные списки вершин для многоугольника отсечения и предметного многоугольника. Ведь наперед известно, что они достаточно простые. На рис. 10.15 проиллюстрирована работа этого упрощенного алгоритма на простом примере,

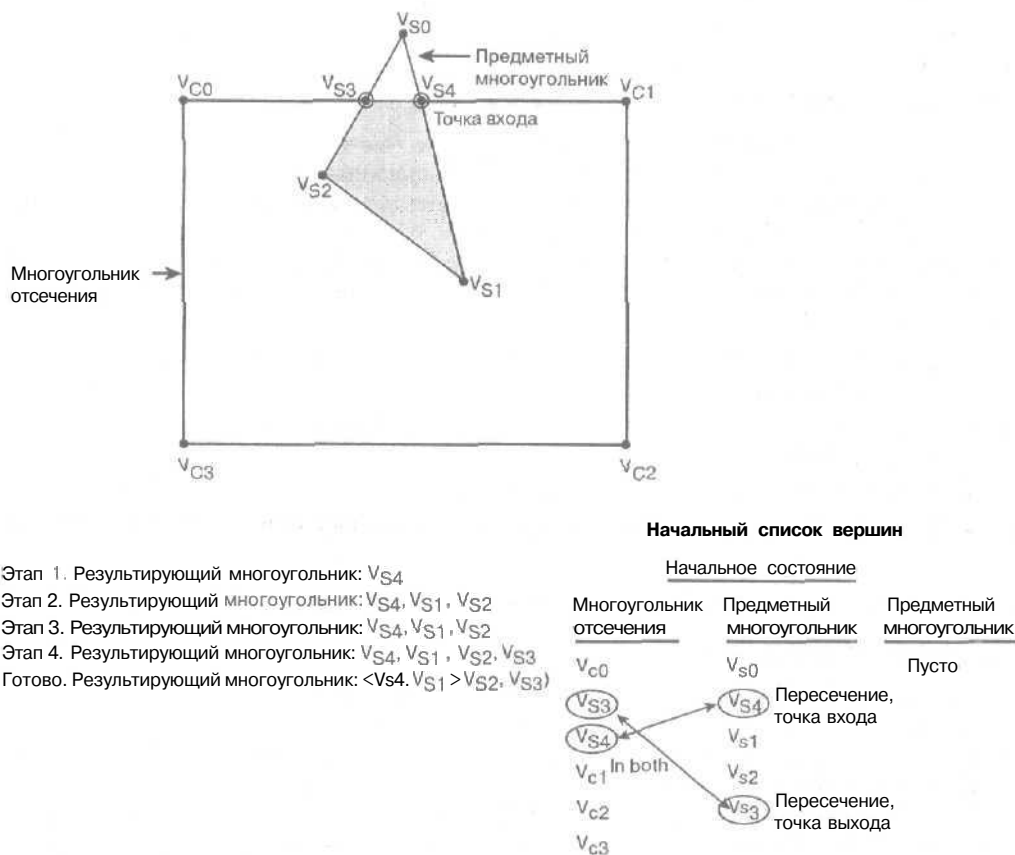


Рис. 10.14. Алгоритм Вейлера-Азертон в действии

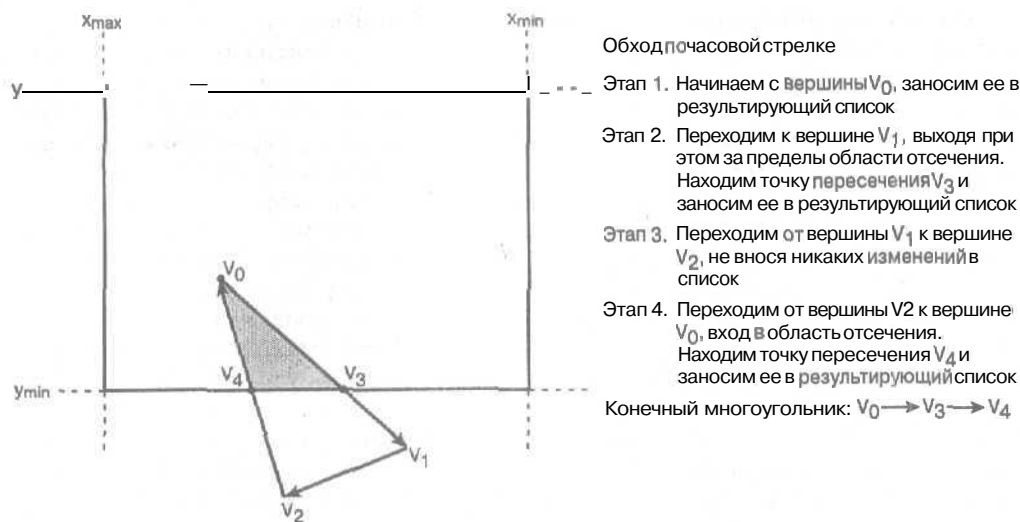


Рис. 10.15. Упрощенный алгоритм Вейлера-Азертонна

## Дальнейшее изучение отсечения

Вот и все. Теперь вы умеете выполнять отсечение. Честно говоря, в этой операции нет ничего сложного — просто применяется один из описанных алгоритмов или их гибрид. В основном все сводится к нахождению пересечения обрабатываемого многоугольника с двумерной или трехмерной областью. Далее мы займемся практической реализацией трехмерного отсечения в разрабатываемом игровом процессоре. Однако если вы хотите больше узнать об этом, есть замечательные книги по данной теме. Некоторые из них представлены ниже.

- Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley
- Foley, van Dam, Feiner, Hughes, and Phillips. *Introduction to Computer Graphics*. Addison-Wesley
- Watt A. *3D Computer Graphics*. Addison-Wesley
- Rogers D. *Procedural Elements for Computer Graphics*, McGraw-Hill

## Практическое отсечение по границам области обзора

Настало время добавить отсечение в наш трехмерный игровой процессор. По сути, на некотором этапе работы процессора нужно выполнить отсечение многоугольников по границам трехмерной области обзора. Как я уже говорил, наш план состоит в том, чтобы проделать эту операцию с минимальными усилиями.

Итак, в каких случаях невозможно обойтись без отсечения? Это происходит только тогда, когда многоугольник выходит за ближнюю плоскость отсечения, или (что еще хуже) простирается в полупространство  $z < 0$ . Проецирование многоугольников, координата  $z$  которых отрицательна или равна нулю, может привести к ошибке деления на ноль и инвертированию проекций координат  $(x, y)$ , что недопустимо.

До **настоящего** времени нам удавалось уходить от решения этой проблемы, отодвигая ближнюю плоскость отсечения на достаточно большое расстояние от плоскостей проекции и  $z = 0$ , чтобы ни один многоугольник не смог одновременно находиться и в области обзора, и достигать этих плоскостей. А те многоугольники, которые находились слишком близко к точке **наблюдения** и плоскости  $z = 0$ , попросту игнорировались. Однако на данном этапе нужно забыть о отбраковке на уровне объекта, и обдумать проблему в терминах общего списка многоугольников, которые могут находиться где угодно.

Мы собираемся разработать некую смесь функций отсечения и отбраковки, в которой производилось бы отсечение по границам области обзора на уровне списка визуализации. Однако на самом деле отсечение в этой функции будет производиться только по ближней плоскости. Многоугольники, которые частично выходят за рамки других границ области обзора, останутся неизменными, а те, что полностью выходят за рамки границ, — просто будут отбракованы. При этом после **проецирования**, конечно же, останутся **многоугольники**, выходящие за рамки двумерного поля обзора, однако их отсечение будет производиться уже на этапе **растеризации**, и они не причинят нам никакого беспокойства. Как уже упоминалось, такая стратегия оказывается более простой и производительной, чем **та**, при которой каждый многоугольник отсекается по всем шести плоскостям, образующим область обзора. В конце концов, такое интенсивное отсечение оборачивается большими затратами времени, поскольку приходится заново разбивать усеченный многоугольник на треугольники, вычислять их нормали и координаты текстуры, и все понапрасну, потому что все равно все это приходится растеризовать! *Единственный* вид многоугольников, для которого *нужно* выполнять отсечение, — это **многоугольники**, которые пересекают ближнюю плоскость и в процессе проецирования могут вызывать искажения и математические ошибки. Замечательная особенность нашего подхода заключается в том, что мы не только собираемся **удалить** обратные поверхности, но и отбраковываем многоугольники, не принадлежащие области обзора. Удастся избежать обработки еще более значительного количества геометрических элементов, чем до **этого**, что позволяет дополнительно повысить производительность программы.

Отсечение трехмерной областью обзора можно выполнять на различных этапах работы трехмерного конвейера: в мировых координатах, в координатах камеры или в аксонометрических координатах. Каждый из этих подходов обладает своими достоинствами и недостатками. Например, отсечение в мировых координатах позволяет избежать **преобразования** всех отсеченных геометрических элементов в систему координат камеры. Однако далее "очищенное" пространство обзора помещается перед точкой наблюдения с произвольной ориентацией. Таким образом, процедура отсечения и отбраковки нуждается в определенном обобщении. С другой стороны, если параметры многоугольников преобразованы из мировой системы координат в систему координат камеры, их можно аккуратно разместить по отношению к камере, которая находится в начале координат, а плоскости, **являющиеся границами** области обзора, расположить параллельно и перпендикулярно осям, благодаря чему процедура отсечения упрощается. При таком подходе затрачивается излишнее время на преобразование всех геометрических элементов из мировой системы координат в систему координат **камеры**, однако сам процесс отсечения и отбраковки упрощается. **Далее**, если подождать до выполнения аксонометрического преобразования (на самом деле все не так просто), то всю геометрию можно нормализовать и привести область обзора к каноническому кубическому виду. Другими словами, если сначала выполнить аксонометрическое преобразование всех геометрических элементов, а **затем** произвести **отсечение**, то отпадает необходимость усекать многоугольники по границам пирамиды, вместо которой теперь у нас имеется куб. Эта особенность неоднократно описана в математических главах, посвященных проецированию трехмерной области.

Что же нам выбрать? О том, чтобы ждать выполнения аксонометрического преобразования, лучше забыть сразу. Так долго тянуть нельзя. При этом нам пришлось бы не только моделировать **освещение** многоугольников, которые вовсе в этом не **нуждаются**, но и перегрузить конвейер излишним количеством многоугольников, которые в итоге не будут визуализированы, а это слишком расточительный подход. Можно было бы выполнить отсечение в мировых координатах, но лично меня не воодушевляет перспектива того, что придется преобразовывать область обзора, помещая ее на нужное место, а затем производя отсечение (хотя это не так уж плохо). Кроме того, возникнет необходимость выполнять преобразование всех дополнительных многоугольников, полученных в результате разбиения усеченных **многоугольников**. Предположим, что всего в поле действия имеется около 1000 многоугольников, а в результате отсечения и разбиения окажется около 1200 (конечно же, здесь я преувеличиваю). А теперь учтите, что все эти 1200 многоугольников нужно будет преобразовывать в систему координат камеры. К тому же проверка, нуждается ли многоугольник в **отсечении**, в мировых координатах несколько сложнее, чем в координатах камеры (которая по определению размещается в начале координат, в результате чего направление на область обзора совпадает с положительным направлением оси  $z$ ).

С другой стороны, **предположим**, что отсечение производится в мировой системе координат, в результате чего удаётся отсеять приблизительно половину многоугольников. Тогда в систему координат камеры нужно будет преобразовать лишь 500 многоугольников. Эти два случая демонстрируют, что каждый подход обладает как своими достоинствами, так и своими недостатками. В конце **концов**, я **решил** выполнять отсечение и отбраковку **многоугольников** после перехода из мировой системы координат в систему координат камеры. Так будет проще для понимания, и я не **думаю**, что так уж много выиграю, производя эти **операции** в мировых координатах. Однако в конце книги, на этапе оптимизации, можно попробовать и другой подход.

Подведем итоги. Отсечение и отбраковку решено выполнять после перехода из мировой системы отсчет в систему координат камеры. Многоугольники, которые полностью выходят за рамки области обзора, будут удалены, а отсекаются будут лишь те, которые пересекают ближнюю плоскость. Пересечение многоугольниками верхней, нижней, правой и левой границ области обзора допускается, поскольку далее предстоит дополнительная их обработка в пространстве изображений на этапе растеризации. Рассмотрим рис. 10.16, где представлена схема конвейера, в который добавлено отсечение.

## Конвейер обработки геометрии и структуры данных

Как видно из рис. 10.16, геометрия игры загружается в систему в виде цельных объектов (или генерируется как набор отдельных многоугольников). После этого происходит преобразование объектов, отбраковка их элементов, преобразование объектов в набор многоугольников, удаление обратных поверхностей, еще одно преобразование, отсечение и **отбраковка**, **освещение**, снова преобразование и отображение на экране. Просто масса преобразований! Дело в том, что мы начали с загрузки объектов, что замечательно работает для тех игр, в которых отсутствуют **интерьерные** и наружные элементы, но плохо для тех игр, в которых эти элементы присутствуют.

Пора перестать придерживаться концепции "объектов" и подумать над новой формой представления геометрии игры, которая бы лучше подошла для моделирования ландшафтов, интерьера помещений и других подобных вещей. Вот почему мы старались использовать как можно меньше функций, выполняющих **освещение** и отбраковку многоугольников на уровне объекта. Предлагаю вам научиться мыслить терминами **много-**

угольников, поскольку скоро нам предстоит освоить более развитые структуры данных, поддерживающие **большие** объекты. Однако не хотелось бы, чтобы эти объекты представляли собой просто большую свалку **многоугольников**. Возможно, было бы удобно, если бы многоугольники были упорядочены, отсортированы, чтобы между ними существовали взаимосвязи, чтобы они подчинялись определенной иерархии и т.д.



Рис. 10.16. Схема конвейера, в который добавлено трехмерное отсечение

В качестве примера рассмотрим рис. 10.17. На нем с высоты птичьего полета представлен вид одного из уровней игры, моделирующей городскую жизнь. Модель этого уровня можно было бы поместить в контейнер, внутреннее устройство которого подобно геометрическому объекту. Однако здесь нужны вторичные структуры данных, позволяющие быстро отбраковывать отдельные геометрические компоненты. В этой **ситуации** приходит на помощь концепция деревьев бинарного разбиения пространства (BSP — binary space partitioning), порталов и т.п. концепции.

Чтобы все это работало, необходимо организовать **эффективный** доступ к различным элементам геометрии, и на определенном этапе преобразовать их в хорошо знакомые **многоугольники**. Я придерживаюсь точки зрения, согласно которой сначала нужно четко представить себе, игры какого типа вы собираетесь разрабатывать. Далее следует осознать, что если это игры с моделированием внешних **ландшафтов** и внутренней обстановки помещений, то отсечение в них играет очень важную роль (**это также касается** отбраковки многоугольников на уровне списка визуализации). Поэтому нужно организовать дело так, чтобы обе эти операции работали эффективно! Вот почему мы решили **добавить** поддержку отсечения и отбраковки в список визуализации, а не перегружать программу выполнением этих **действий** на уровне объекта. У нас уже просто не получается приспособливать структуры данных, в которых хранятся параметры объекта, для **всех** операций.

## Добавление отсечения в игровой процессор

Начнем сначала. Новый библиотечный модуль, содержащий код отсечения, получил имя `T3DLIB8.CPP`. Он очень короткий и содержит всего несколько функций. Как обычно, хотелось бы представить краткий обзор этого библиотечного модуля, не вдаваясь в объяснения, почему в него помещена та или иная функция. Итак, рассмотрим заголовочный файл.

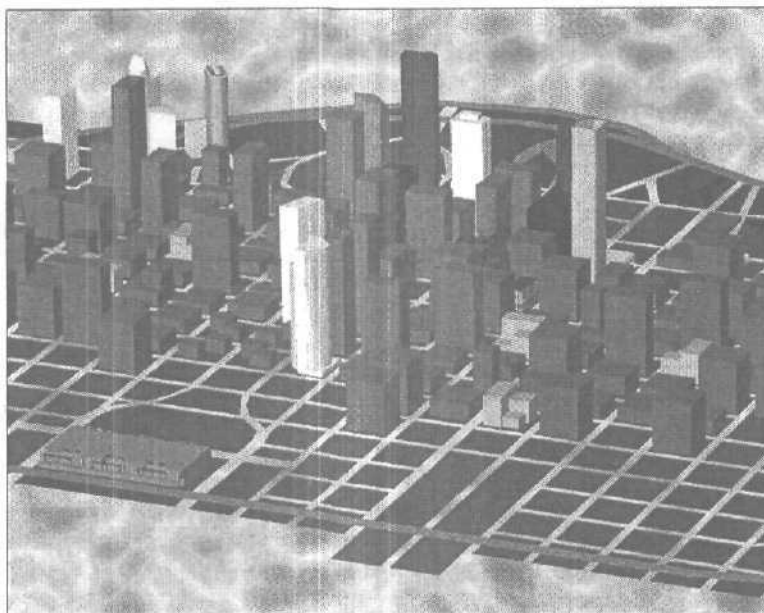


Рис. 10.17. Вид игрового уровня, моделирующего городской ландшафт

### Заголовочный файл

В функции отсечения используется всего несколько констант. Все они имеют отношение к управлению логикой работы этой функции и определяют, по каким плоскостям производится отсечение. Приведем их определения.

```
// Общие флаги отсечения для многоугольников
#define CLIP_POLY_X_PLANE 0x0001 // Отбраковка по
// плоскостям отсечения x
#define CLIP_POLY_Y_PLANE 0x0002 // Отбраковка по
// плоскостям отсечения y
#define CLIP_POLY_Z_PLANE 0x0004 // Отбраковка по
// плоскостям отсечения z
#define CLIP_OBJECT_XYZ_PLANES (CULL_OBJECT_X_PLANE | \
CULL_OBJECT_Y_PLANE | \
CULL_OBJECT_Z_PLANE)
```

В этой библиотеке всего две функции. Одна из них производит **отсечение**, а другая — генерирует ландшафт (это сюрприз).

```
void Clip_Polys_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, // Список визуализации
    CAM4DV1_PTR cam, // Камера
    int clip_flags); // Флаги отсечения

int Generate_Terrain_OBJECT4DV2(
    OBJECT4DV2_PTR obj, // Указатель на объект
    float width, // Ширина по оси x в мировой
// системе координат
    float height, // Длина по оси z в мировой
```

```

float vscale,           // системе координат
                        // Масштаб ландшафта по
                        // вертикали
char *height_map_file, // Имя файла с битовым образом,
                        // закодированным в 256-
                        // цветовом формате
char *texture_map_file, // Имя файла с текстурной
                        // картой
int rgbcolor,           // Цвет ландшафта при
                        // отсутствии текстуры
VECTOR4D_PTR pos,       // Начальное положение
VECTOR4D_PTR rot        // Начальная ориентация
int poly_attr);         // Атрибуты затенения

```

Пока все довольно просто; теперь обсудим, как работает функция отсечения.

## Разработка функции отсечения

Функция отсечения должна работать в системе координат камеры на уровне списка визуализации. Это означает, что точка наблюдения расположена в начале координат, а область отсечения находится в полупространстве с положительными координатами  $z$  (рис. 10.18).

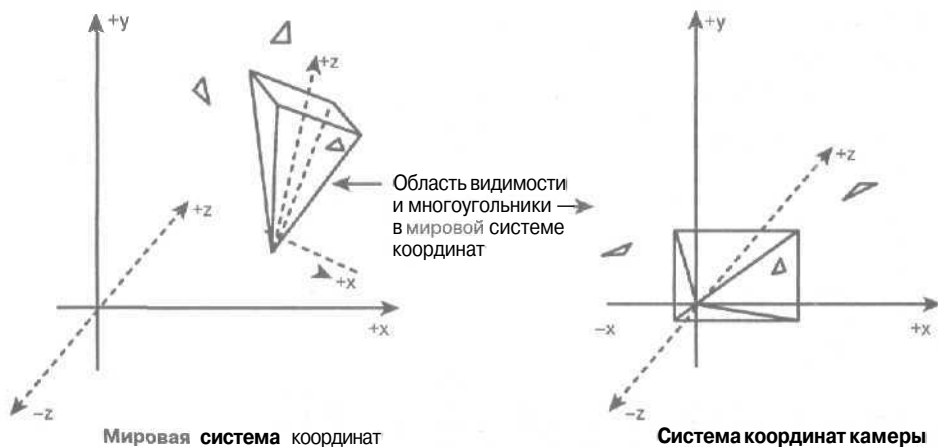


Рис. 10.18. Отсечение в системе координат камеры

Ниже приведен стандартный листинг, из которого видно, какие операции производятся в текущем конвейере для того, чтобы выполнить визуализацию трехмерных объектов.

```

Reset_OBJECT4DV2(&obj);

// Создание единичной матрицы
MAT_IDENTITY_4X4(&mtrans);

// Преобразование каркаса
Transform_OBJECT4DV2(&obj, &mtrans,
                    TRANSFORM_LOCAL_TO_TRANS,1);

// Преобразование в мировую систему координат
Model_To_World_OBJECT4DV2(&obj, TRANSFORM_TRANS_ONLY);

```

```

// Помещение объекта в список визуализации
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &obj, 0);

// Удаление обратных поверхностей
Remove_Backfaces_RENDERLIST4DV2(&rend_list, &cam);

// Переход из мировой системы координат
// в систему координат камеры
World_To_Camera_RENDERLIST4DV2(&rend_list, &cam);

////////////////////////////////////
// Здесь будет производиться отсечение многоугольников //
// по границам области обзора //
////////////////////////////////////

// Освещение места действия
Light_RENDERLIST4DV2_World16(&rend_list, &cam, lights, 4);

// Сортировка многоугольников в списке (быстрая!)
Sort_RENDERLIST4DV2(&rend_list, SORT_POLYLIST_AVGZ);

// Переход из системы координат камеры в аксонометрическую
// систему координат
Camera_To_Perspective_RENDERLIST4DV2(&rend_list, &cam);

// Переход в экранную систему координат
Perspective_To_Screen_RENDERLIST4DV2(&rend_list, &cam);

// Блокировка заднего буфера
DDraw_Lock_Back_Surface();

// Визуализация многоугольников
Draw_RENDERLIST4DV2_Solid16(&rend_list, back_buffer,
                             back_lpitch);

// Разблокирование заднего буфера
DDraw_Unlock_Back_Surface()

```

В приведенном листинге указано, в каком месте игрового конвейера будет производиться отсечение многоугольников: после перехода в систему координат камеры и перед моделированием освещения. Это идеальное место для такой операции, поскольку после отсечения и отбраковки лишних многоугольников их не только становится меньше, но и те из них, которые были удалены из списка **визуализации**, уже не нужно **освещать**.

Теперь поговорим о том, что нужно сделать, чтобы получить возможность выполнять отсечение на уровне списка визуализации. Этот список состоит из двух структур данных: массива многоугольников и массива указателей на них. Необходимость иметь в своем распоряжении указатели объясняется тем, что их сортировка выполняется быстрее, и при этом даже не надо выделять дополнительные области памяти или удалять их, поскольку основное **хранилище** информации всегда в нашем распоряжении.

#### **Отсечение по верхней, нижней, правой и левой границам области обзора**

Мы пришли к выводу, что не будем производить отсечение многоугольников по верхней, нижней, правой и левой границам области обзора. Мы будем определять, находится ли обрабатываемый многоугольник в области обзора или полностью выходит за ее пределы. Если многоугольник попадает в область обзора, он направляется для дальнейшей обработки в игровой конвейер, и мы переходим к проверке следующего

многоугольника. Если же многоугольник **полностью** выходит за плоскость, он отбраковывается (рис. 10.19).

Чтобы выполнить поставленную задачу, достаточно знать координаты вершин тестируемого треугольника, а также положение и ориентацию самих плоскостей отсечения. Эти сведения можно получить несколькими способами. Если бы отсечение производилось в мировых координатах, удобнее всего было бы использовать уравнение плоскости, заданное с **помощью** точки, **принадлежащей** этой плоскости, и нормали к ней. Затем с помощью скалярного произведения можно было бы узнать, в каком полупространстве по отношению к данной плоскости находится та или иная вершина. Если все три **вершины** треугольника находятся в одном и том же полупространстве, треугольник отбраковывается или принимается для дальнейшей обработки.

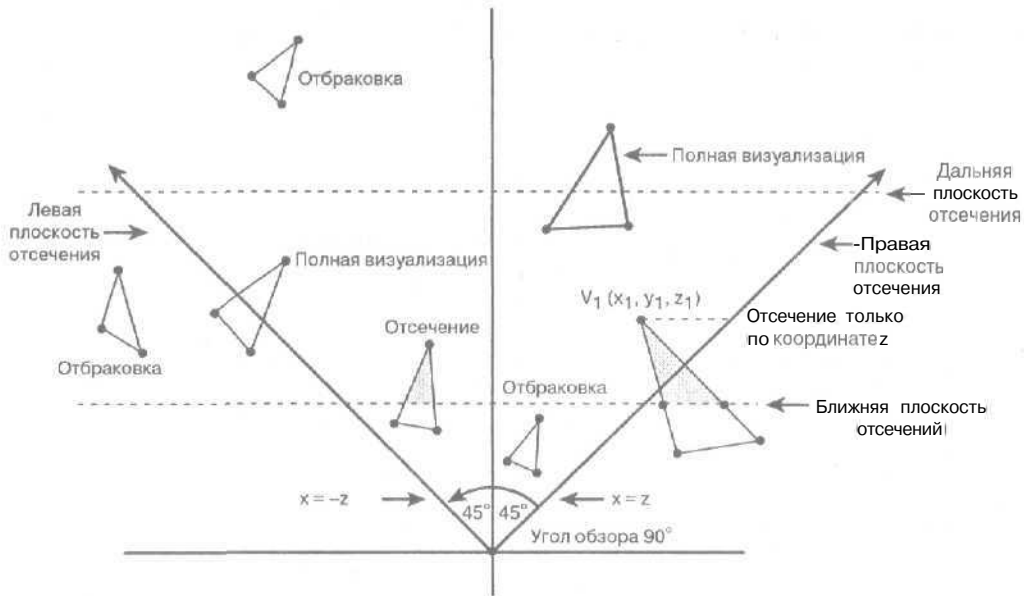


Рис. 10.19. Отсечение с нестрогим соблюдением границ

Однако мы можем несколько упростить процедуру, поскольку заранее известно, что плоскости отсечения перпендикулярны или параллельны некоторым осям системы координат камеры. Это позволяет использовать их наклон по отношению к другим осям (угол обзора) для определения того, находится ли точка в области обзора. Еще раз обратите внимание на рис. 10.19. Допустим, нужно определить положение многоугольника  $l$  относительно правой границы области обзора. На время можно забыть о компоненте  $y$  и рассуждать только в терминах плоскости  $xz$ . Вопрос стоит так: где по отношению к правой плоскости отсечения находится вершина  $v_1(x_1, y_1, z_1)$ ? Чтобы ответить на него, воспользуемся свойствами подобных треугольников.

На рис. 10.19 использован угол обзора, равный  $90^\circ$ . Это означает, что и **правая**, и **левая** границы области обзора имеют наклон к оси  $z$ , равный  $45^\circ$ , а уравнение правой плоскости имеет вид  $x = z$  (координата  $y$  может быть произвольной). Если для тестируемой вершины  $v_1$  выполняется условие  $x_1 = z_1$ , то она принадлежит плоскости отсечения; если  $x_1 > z_1$ , то вершина находится в отрицательном полупространстве по отношению к **этой**

плоскости (по другую сторону от области обзора), а если  $x_1 < z_1$ , — то в положительном полупространстве (по ту сторону, где расположена область обзора).

В общем случае описанная выше проверка для вершины  $v(x, y, z)$  и угла обзора  $90^\circ$  имеет следующий вид:

- если  $x > z$ , вершина  $v$  находится за правой отсекающей плоскостью;
- если  $x < z$ , вершина  $v$  находится перед правой отсекающей плоскостью;
- если  $x = z$ , вершина  $v$  принадлежит правой отсекающей плоскости.

Аналогичную проверку можно выполнить для левой границы области обзора:

- если  $-x > z$ , вершина  $v$  находится за левой отсекающей плоскостью;
- если  $-x < z$ , вершина  $v$  находится перед левой отсекающей плоскостью;
- если  $-x = z$ , вершина  $v$  принадлежит левой отсекающей плоскости.

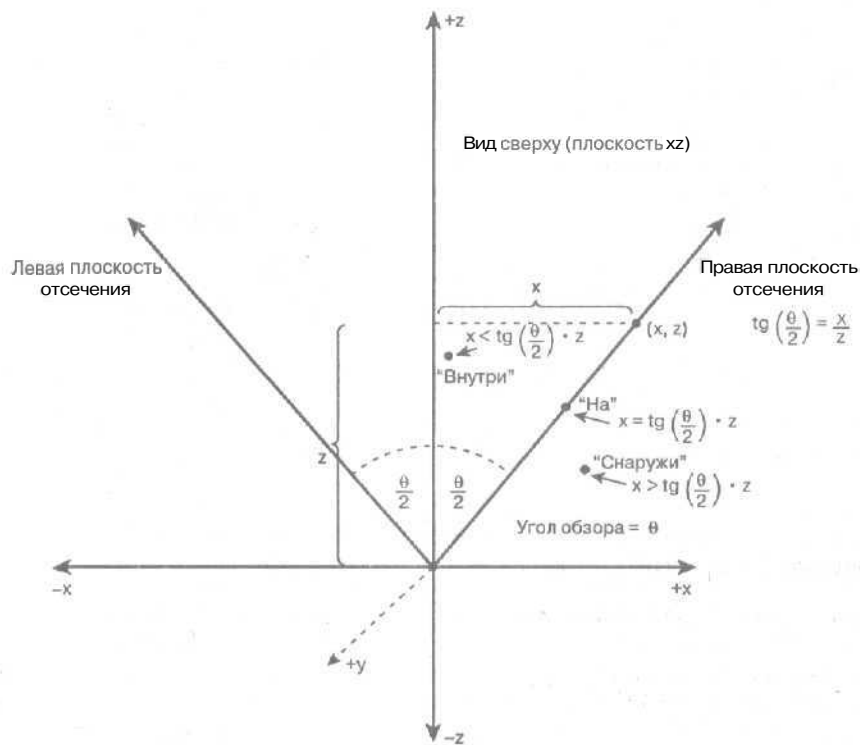


Рис. 10.20. Определение положения вершины на основе наклона плоскости отсечения

Аналогичный процесс нужно выполнить для верхней и нижней границ; при этом просто вместо координаты  $x$  будет использоваться координата  $y$ . Конечно же, в этом примере рассматривается частный случай, и описанный в нем способ работает, только если угол обзора равен  $90^\circ$ . Однако изменение угла обзора сводится к изменению наклона плоскостей сечения к оси  $z$ , поэтому для учета угла обзора достаточно лишь немного изменить формулы. На рис. 10.20 приведен общий случай для правой и левой плоскостей сечения с произвольным наклоном. Из рисунка видно, что при проверке следует использовать тот факт, что

$$\operatorname{tg} \frac{\theta}{2} = \frac{x}{z}, \text{ т.е. } x = z \cdot \operatorname{tg}(\theta/2)$$

Другими словами, мы получаем следующие ситуации:

- если  $|x| = z \cdot \operatorname{tg}(\theta/2)$ , тестируемая вершина *принадлежит* плоскости отсечения;
- если  $|x| < z \cdot \operatorname{tg}(\theta/2)$ , тестируемая вершина находится в *положительном полупространстве* по отношению к плоскости отсечения (по одну сторону с областью обзора);
- если  $|x| > z \cdot \operatorname{tg}(\theta/2)$ , тестируемая вершина находится в *отрицательном полупространстве* по отношению к плоскости отсечения (по другую сторону от области обзора).

Суть в том, что для каждой вершины необходимо определить, находится ли она в положительных полупространствах по отношению ко всем четырём плоскостям отсечения (верхней, нижней, правой и левой). Это означает, что нужно выполнить сравнение координат  $x$  и  $y$  вершины  $v(x, y, z)$  с её координатой  $z$  с учетом угла обзора, тангенс половины которого будет выступать в роли множителя в условиях сравнения.

Подобные рассуждения уже приводились сотни раз, поэтому у вас не должно возникнуть трудностей с пониманием **того**, о чем здесь идет речь. Как бы там ни было, если все вершины треугольника находятся в области обзора, мы просто переходим к следующему треугольнику. Если же все три вершины выходят за рамки области обзора, то треугольник полностью отбраковывается и помечается как "отсеченный".

```
// Полностью выбрасываем многоугольник из области обзора
SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);
```

Чтобы сэкономить место, рассмотрение листинга будет отложено до того момента, когда мы перейдем к обсуждению отсечения по оси  $z$ . Однако для краткого ознакомления с реализацией разработанных выше тестов приведем небольшие фрагменты кода.

```
// Поскольку отсечение производится по правой и левой
// границам области обзора, нужно с помощью параметров
// поля зрения или уравнения этих плоскостей определить,
// при каком соотношении между координатами  $x$  и  $z$  точка
// находится во внешней части по отношению к плоскости
```

```
z_factor = (0.5)*cam->viewplane_width/cam->view_dist;
```

```
// Вершина O
z_test = z_factor*curr_poly->tvlist[0].z;
```

```
if (curr_poly->tvlist[0].x > z_test)
    vertex_ccodes[0] = CLIP_CODE_GX;
else
    if (curr_poly->tvlist[0].x < -z_test)
        vertex_ccodes[0] = CLIP_CODE_LX;
    else
        vertex_ccodes[0] = CLIP_CODE_IX;
```

Такое подробное объяснение всего для десятка строк кода! Но как я уже говорил, нам всего лишь нужно знать, по какую сторону от плоскости отсечения находится вершина. Основываясь на этих сведениях, в коде устанавливаются те или иные флаги отсечения

(аналогично тому, как это делается в алгоритме Кохена-Сазерленда), позволяющие отслеживать для каждой вершины, находится ли она в области обзора или вне нее. Это позволяет в дальнейшем направлять многоугольники для дальнейшей обработки или выключать их из рассмотрения. Теперь пора перейти к обсуждению действий алгоритма, производимых для ближней и дальней плоскостей отсечения, перпендикулярных оси  $z$ , потому что в этой части все происходит несколько сложнее.

#### Отсечение по ближней и дальней плоскостям отсечения

Здесь все несколько усложняется. Говорить-то об отсечении легко и просто, а вот выполнить его, да еще в трехмерном пространстве... Это связано не со сложностью алгоритма или математических выражений, а из-за изматывающих деталей реализации, в чем вы скоро убедитесь. Итак, предположим, что отбракованы все многоугольники, выходящие за рамки правой, левой, верхней и нижней плоскостей отсечения. Для оставшихся многоугольников нужно проверить, какое положение они занимают по отношению к ближней и дальней плоскостям отсечения. Эти проверки **тривиальны** — для каждой вершины  $v_i(x_i, y_i, z_i)$  (индекс  $i$  изменяется от 0 до 2) тестируемого многоугольника достаточно сравнить ее координату  $z$  со значениями  $z$ , задающими положение плоскостей отсечения. В структуре данных, в которой хранятся параметры камеры, эти переменные имеют имена `near_clip_z` и `far_clip_z`, соответственно.

Например, чтобы проверить, выходит ли треугольник полностью за пределы дальней плоскости отсечения, нужно выполнить следующий псевдокод.

```
if ((v0.z > far_clip_z) &&
    (v1.z > far_clip_z) &&
    (v2.z > far_clip_z))
{
    Многоугольник (v0, v1, v2) отбраковывается
}
```

Просто, правда? Аналогичный тест можно выполнить и для ближней плоскости отсечения.

```
if ((v0.z < near_clip_z) &&
    (v1.z < near_clip_z) &&
    (v2.z < near_clip_z))
{
    Многоугольник (v0, v1, v2) отбраковывается
}
```

Нас не интересуют случаи, когда треугольники лишь частично находятся в области обзора, если они выходят за пределы дальней плоскости отсечения. Однако если треугольник пересекает ближнюю плоскость, это *важно*. Это проблема, избежать которой нельзя. Мы собираемся решить ее следующим образом.

```
for каждый многоугольник P из списка визуализации
begin
    for каждая вершина v текущего многоугольника P
    begin
        Записываем положение вершины v:
        1. Находится ли она в области обзора?
        2. Выходит ли она за пределы дальней плоскости?
        3. Выходит ли она за пределы ближней плоскости?
    end
```

Если все три вершины выходят за пределы ближней или

дальней плоскостей отсечения, то многоугольник Р полностью отбраковывается

**Если (некоторые вершины находятся в области обзора) и (некоторые вершины *выходят* за рамки ближней плоскости) то выполняем отсечение** многоугольника Р по ближней плоскости отсечения.

end

Теперь нам нужно реализовать часть псевдокода, выделенную полужирным шрифтом. Проблема в том, что для этого необходимо понять, как производить отсечение многоугольника ближней границей области обзора. Этот процесс в основном сводится к отсечению этой плоскостью каждой стороны многоугольника, пересекающей эту плоскость. Рассмотрим два случая, представленных на рис. 10.21.

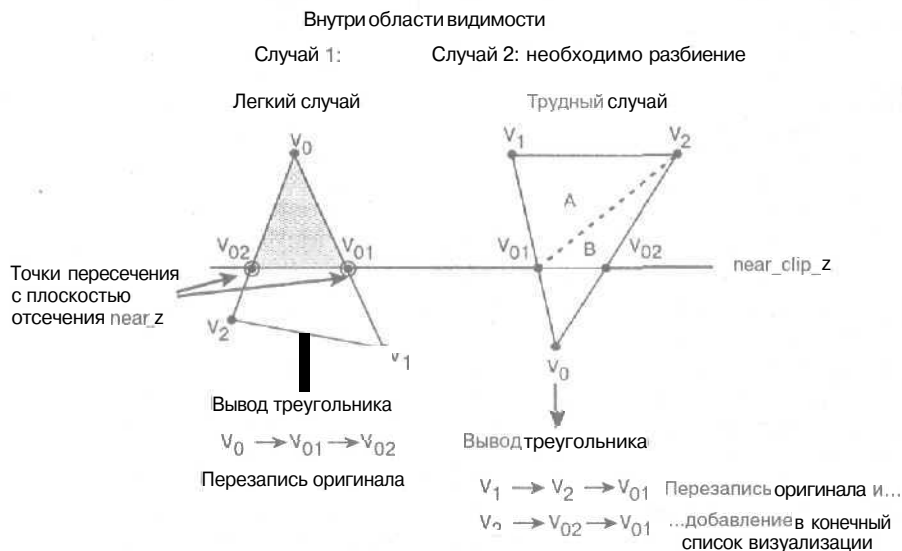


Рис. 10.21. Два случая отсечения по ближней границе области обзора

#### Случай 1: одна внутренняя вершина и две внешние

Это более простой случай. В нем производится обычное отсечение двух сторон, выходящих за пределы ближней плоскости, а затем информация о новых вершинах перезаписывается вместо старых. Кроме того, если на треугольник накладывается текстура, нужно заново вычислить координаты текстуры. Конечно, нормаль также следует вычислить повторно, поскольку ее длина изменится в результате уменьшения площади треугольника. Кроме того, чтобы ускорить вычисления, связанные с моделированием освещения, нужно сохранить новую длину нормали (которую также нужно вычислить повторно, поскольку она изменилась).

#### Случай 2: две внутренние вершины и одна внешняя

Этот случай сложнее, чем предыдущий. Снова обратимся к рис. 10.21, к той его части, где две вершины треугольника находятся внутри области обзора, а одна — вне этой области. В этом случае также есть две точки пересечения сторон многоугольника с ближней плоскостью отсечения. Полученный в результате усечения многоугольник является че-

тырехугольником, с которым игровой процессор работать не в состоянии. Таким образом, его нужно разбить на треугольники! Используемая стратегия заключается в том, чтобы скопировать всю информацию о треугольнике во временное хранилище, выполнить его отсечение и разбиение на два треугольника, А и В. Затем параметры треугольника А заносятся туда, где ранее хранилась информация об исходном треугольнике, в результате чего одна из точек пересечения становится его вершиной. В итоге получается, что исходный многоугольник в списке визуализации модифицируется, т.е. вместо него записывается один из треугольников, образовавшийся при разбиении. Кроме того, в список необходимо поместить другой треугольник (треугольник В).

Прежде чем продолжить, еще раз убедимся в том, что хорошо поняли случай 2. Суть в том, что после отсечения треугольника плоскостью вместо него может образоваться один или два новых треугольника. Если оказалось, что треугольник один, информация о нем записывается туда, где хранились параметры старого треугольника, что довольно безболезненно. Если же в результате отсечения треугольника получится четырехугольник, который разбивается на два треугольника (А и В), то исходный треугольник перезаписывается, однако в список визуализации необходимо добавить еще один треугольник.

#### СОВЕТ

Можно заметить, что было бы лучше, если бы у нас был связанный список многоугольников, а не список указателей на них. Дело в том, что треугольники, получившиеся в результате разбиения, при сортировке окажутся в конце списка многоугольников. Это ясно из того, что многоугольники сортируются в порядке убывания координаты z. В результате сортировка замедляется, поскольку многоугольники отодвигаются в самый конец списка. Однако в дальнейшем будет произведена оптимизация, устраняющая эту аномалию.

Теперь, когда план действий разработан, осталась последняя деталь — разработать сам код отсечения. Как его реализовать? Я остановился на алгоритме, представляющем собой гибрид описанных ранее алгоритмов. Этап предварительного кодирования входных и выходных величин уже обсуждался. Единственная оставшаяся проблема состоит в том, чтобы определить, как производить отсечение сторон треугольника по ближней плоскости отсечения. Поскольку заранее известно, что существует всего два различных случая (когда внутри области обзора находится одна или две вершины треугольника), я решил составить для каждого из них отдельный код. В каждом частном случае я создаю параметрические уравнения прямых, на которых лежат отсекаемые стороны. Они имеют примерно такой вид:

$$p = v0 + (v1 - v0) * t$$

где  $v0$  и  $v1$  — концы отрезка, который является стороной треугольника. В компонентном представлении это уравнение имеет вид:

$$\begin{aligned} x &= v0.x + (v1.x - v0.x) * t \\ y &= v0.y + (v1.y - v0.y) * t \\ z &= v0.z + (v1.z - v0.z) * t \end{aligned}$$

Далее в последнее уравнение вместо координаты z подставляется значение `near_clip_z`, и оно решается относительно параметра t:

$$near\_clip\_z = v0.z + (v1.z - v0.z) * t$$

$$t = (near\_clip\_z - v0.z) / (v1.z - v0.z)$$

Затем полученное значение t подставляется в уравнения для компонент x и y, и мы находим точку пересечения. Выполняем эту операцию для обеих сторон — и все готово. Остальное — дело техники, хотя следует позаботиться о том, чтобы перезаписать информацию о вершине данными, соответствующими точке пересечения.

### Вычисление координат текстуры

Единственное, что осталось сделать, — вычислить новые координаты текстуры. Здесь нужно соблюдать осторожность. Если на многоугольник накладывается текстура, то ее координаты также нужно отсекать. К счастью, для этого можно использовать уже найденное значение параметра  $t$ . Рассмотрим пример. Предположим, имеется треугольник первого типа (рис. 10.22), в каждой вершине которого заданы координаты текстуры, обозначенные так, как показано на рисунке.

Предположим, что пересечение происходит при значении параметра  $t$ , равном 0.6. При линейной интерполяции координат текстуры используется то же значение параметра  $t$ . Как видно из рисунка, вершине 0 соответствуют координаты текстуры  $(u_0, v_0) = (3, 5)$ , а вершине 1 — координаты текстуры  $(u_1, v_1) = (15, 50)$ . Таким образом, отсеченные координаты текстуры равны:

$$u_{\text{clipped}} = u_0 + (u_1 - u_0) \cdot t$$

$$v_{\text{clipped}} = v_0 + (v_1 - v_0) \cdot t$$

Вычисляя эти величины, получим:

$$u_{\text{clipped}} = 3 + (15 - 3) \cdot (0.6) = 10.2$$

$$v_{\text{clipped}} = 5 + (50 - 5) \cdot (0.6) = 32$$

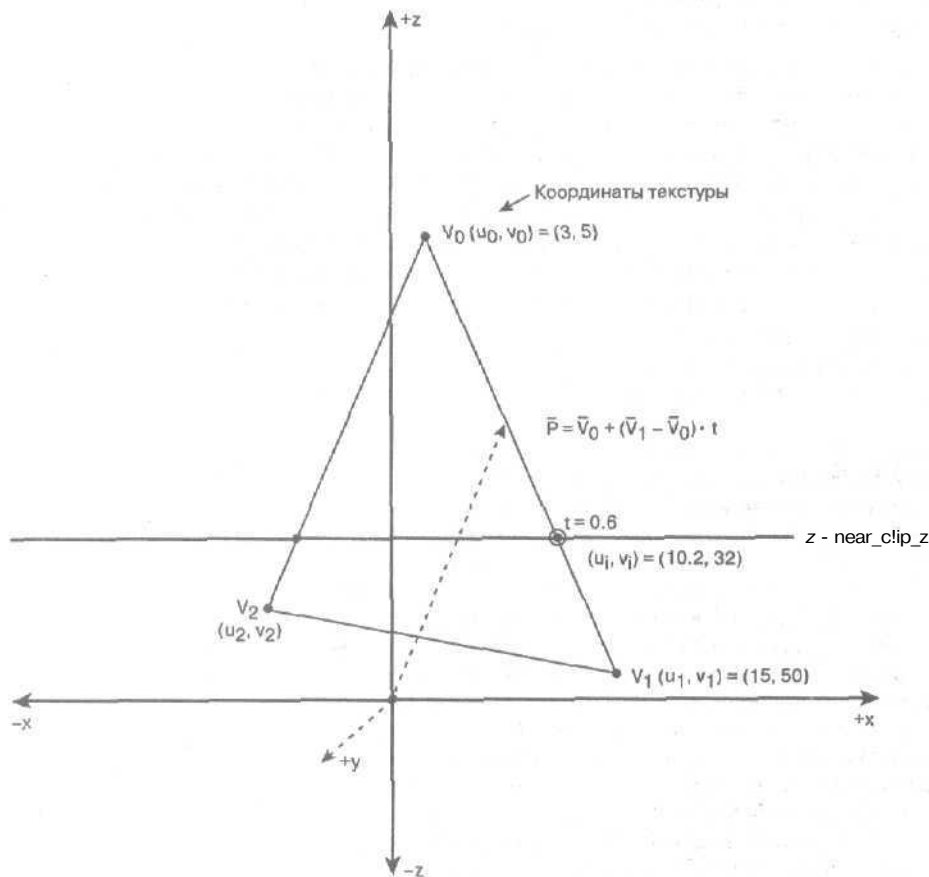


Рис. 10.22. Вычисление новых координат текстуры для усеченного треугольника

Вот и все, что следует знать об отсечении ближней плоскостью с учетом последующего разбиения усеченного многоугольника, а также о повторном вычислении координат текстуры (если это *необходимо*). Наконец, перед тем как выйти на финишную прямую, нужно повторно вычислить длину нормали к каждому усеченному многоугольнику, поскольку она изменилась в результате усечения. Новая длина нормали понадобится при моделировании освещения. Все усеченные многоугольники, независимо от того, пришлось ли их заново разбивать на треугольники или нет, обрабатываются с помощью приведенного ниже *кода*.

```
// Построение и и v
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
               &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
               &curr_poly->tvlist[v2].v, &v);

// Вычисление векторного произведения
VECTOR4D_Cross(&u, &v, &n);

// Быстрое вычисление длины нормали и сохранение
// ее в переменной-члене nlength
curr_poly->nlength = VECTOR4D_Length_Fast(&n);
```

Этот код составлен несколько грубо, однако я надеюсь, что по ближней границе области обзора нужно будет отсекал не более нескольких сотен многоугольников. Поэтому в приведенном фрагменте используется процедура для быстрого вычисления длины. Она немного *неточная*, зато работает намного быстрее, чем та, в которой используется квадратный корень. Относительная погрешность составляет около 5–10%, однако это не должно нас волновать, поскольку, если многоугольники отсекаются по ближней границе области обзора, они, скорее всего, скоро исчезнут из поля зрения.

Надеюсь, еще одна тема рассмотрена до мельчайших подробностей, и пора, наконец, взглянуть на самую получившуюся *функцию* — ее код приведен ниже.

```
void Clip_Polys_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list,
    CAM4DV1_PTR cam, int clip_flags)
{
    // Функция производит отсечение и отбраковку многоугольников
    // из списка визуализации по отношению к указанным
    // плоскостям и устанавливает на отбракованные
    // многоугольники соответствующий флаг, препятствующий их
    // визуализации. Заметим, что отсечение выполняется только
    // ближней и дальней границами области обзора. По отношению
    // к правой, левой, верхней и нижней границам производится
    // тривиальная отбраковка многоугольников, которые полностью
    // выходят за рамки области. Подобный метод неэффективен на
    // уровне объектов, поскольку получается, что видимый объект
    // полностью состоит из видимых многоугольников. Если же из
    // большого объекта сформирован список визуализации, то
    // видимой в нем будет всегда лишь часть многоугольников.
    // Таким образом, тест на видимость лучше выполнять на
    // уровне списка визуализации. В функции предполагается, что
    // многоугольники заданы в системе координат камеры
```

```

// Внутренние коды отсечения.
#define CLIP_CODE_GZ 0x0001 // z > z_max
#define CLIP_CODE_LZ 0x0002 // z < z_min
#define CLIP_CODE_IZ 0x0004 // z_min < z < z_max

#define CLIP_CODE_GX 0x0001 // x > x_max
#define CLIP_CODE_LX 0x0002 // x < x_min
#define CLIP_CODE_IX 0x0004 // x_min < x < x_max

#define CLIP_CODE_GY 0x0001 // y > y_max
#define CLIP_CODE_LY 0x0002 // y < y_min
#define CLIP_CODE_IY 0x0004 // y_min < y < y_max

#define CLIP_CODE_NULL 0x0000

int vertex_ccodes[3]; // Хранилище флагов отсечения
int num_verts_in;    // Количество внутренних вершин
int v0, v1, v2;      // Индексы вершин

float z_factor,      // Используются при вычислениях,
z_test;             // связанных с отсечением

float xi, yi, x01i, y01i, x02i, y02i, // Точки пересечения
t1, t2, // Значения параметра t
ui, vi, u01i, v01i, u02i, v02i; // Координаты текстуры
// в точках пересечения

int last_poly_index, // Последний корректный
// многоугольник в списке
insert_poly_index; // Текущий индекс, с которым
// вставляется новый многоугольник

VECTOR4D u,v,n; // Используются при вычислении
// координат векторов

POLYF4DV2 temp_poly; // Используется, если многоугольник
// разбивается на 2 треугольника

// Присвоение текущему индексу вставляемого многоугольника
// последнего номера в списке. Мы не хотим усекать
// многоугольники по два раза
insert_poly_index = last_poly_index = rend_list->num_polys;

// Начало цикла по всем многоугольникам в списке, в котором
// производится их усечение и отбраковка
for (int poly = 0; poly < last_poly_index; poly++)
{
    // Запрос текущего многоугольника.
    POLYF4DV2_PTR curr_poly = rend_list->poly_ptrs[poly];

    // Корректен ли данный многоугольник?
    // Многоугольник тестируется только в том случае, если он

```

```

// не усекался, не отбраковывался, является активным и
// видимым, а также если он не двусторонний. Обратите
// внимание, что проверка, входе которой устанавливается,
// является ли поверхность объекта обратной, выполняется
// лишь тогда, если при предыдущем вызове это не было
// установлено. Мы стараемся избегать лишней работы
if ((curr_poly==NULL) ||
    !(curr_poly->state & POLY4DV2_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV2_STATE_CLIPPED)||
    (curr_poly->state & POLY4DV2_STATE_BACKFACE))
continue; // Переход к следующему многоугольнику

// Отбраковка по x-плоскостям
if (clip_flags & CLIP_POLY_X_PLANE)
{
    // Отбраковка на основе положения многоугольника по
    // отношению к плоскостям отсечения x. Для каждой
    // вершины определяем, находится ли она по одну
    // сторону с областью обзора или по разные, и в
    // зависимости от этого присваиваем вершине
    // соответствующий код. Треугольники не отсекаются.
    // Некоторые из них просто отбрасываются. Те, что
    // частично выходят за рамки области обзора, будут
    // усечены функцией растеризации по границам
    // прямоугольной области экрана, а те, которые
    // полностью выходят за пределы области обзора -
    // исключаются из рассмотрения. Поскольку отсечение
    // производится по левой и правой границам области
    // обзора, нужно использовать параметры этой
    // области или уравнения плоскостей, чтобы определить,
    // при каких соотношениях между координатами z и x
    // точка лежит за пределами плоскости
    z_factor = (0.5)*cam->viewplane_width/cam->view_dist;

    // Вершина 0

    z_test = z_factor*curr_poly->tvlist[0].z;

    if (curr_poly->tvlist[0].x > z_test)
        vertex_ccodes[0] = CLIP_CODE_GX;
    else
    if (curr_poly->tvlist[0].x < -z_test)
        vertex_ccodes[0] = CLIP_CODE_LX;
    else
        vertex_ccodes[0] = CLIP_CODE_IX;

    // Вершина 1

    z_test = z_factor*curr_poly->tvlist[1].z;

    if (curr_poly->tvlist[1].x > z_test)
        vertex_ccodes[1] = CLIP_CODE_GX;
    else

```

```

if (curr_poly->tvlist[1].x < -z_test)
    vertex_ccodes[1] = CLIP_CODE_LX;
else
    vertex_ccodes[1] = CLIP_CODE_IX;

// Вершина 2

z_test = z_factor*curr_poly->tvlist[2].z;

if (curr_poly->tvlist[2].x > z_test)
    vertex_ccodes[2] = CLIP_CODE_GX;
else
    if (curr_poly->tvlist[2].x < -z_test)
        vertex_ccodes[2] = CLIP_CODE_LX;
    else
        vertex_ccodes[2] = CLIP_CODE_IX;

// Проверяем, можно ли многоугольник исключить из
// рассмотрения. Для этого он должен полностью
// выходить за пределы правой или левой границ
// области обзора.
if (((vertex_ccodes[0] == CLIP_CODE_GX) &&
    (vertex_ccodes[1] == CLIP_CODE_GX) &&
    (vertex_ccodes[2] == CLIP_CODE_GX)) ||

    ((vertex_ccodes[0] == CLIP_CODE_LX) &&
    (vertex_ccodes[1] == CLIP_CODE_LX) &&
    (vertex_ccodes[2] == CLIP_CODE_LX)))

    !
    // Помечаем многоугольник, полностью выходящий за
    // рамки области обзора.
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

// Переходим к следующему многоугольнику.
continue;
} // if

} // if

// Отбраковка по y-плоскостям.
if (clip_flags & CLIP_POLY_Y_PLANE)
{
    // Отбраковка на основе положения многоугольника по
    // отношению к плоскостям отсечения y. Для каждой
    // вершины определяем, находится ли она по одну
    // сторону с областью обзора или по разные, и в
    // зависимости от этого присваиваем вершине
    // соответствующий код. Треугольники не отсекаются.
    // Некоторые из них просто отбрасываются. Те, что
    // частично выходят за рамки области обзора, будут
    // усечены функцией растеризации по границам
    // прямоугольной области экрана, а те, которые

```

```
// полностью выходят за пределы области обзора, -
// исключаются из рассмотрения. Поскольку отсечение
// производится по левой и правой границам области
// обзора, нужно использовать параметры этой
// области или уравнения плоскостей, чтобы определить,
// при каких соотношениях между координатами z и y
// точка лежит за пределами плоскости
z_factor = (0.5)*cam->viewplane_width/cam->view_dist;
```

```
// Вершина 0
```

```
z_test = z_factor*curr_poly->tvlist[0].z;
```

```
if (curr_poly->tvlist[0].y > z_test)
    vertex_ccodes[0] = CLIP_CODE_GY;
else
if (curr_poly->tvlist[0].y < -z_test)
    vertex_ccodes[0] = CLIP_CODE_LY;
else
    vertex_ccodes[0] = CLIP_CODE_IY;
```

```
// Вершина 1
```

```
z_test = z_factor*curr_poly->tvlist[1].z;
```

```
if (curr_poly->tvlist[1].y > z_test)
    vertex_ccodes[1] = CLIP_CODE_GY;
else
if (curr_poly->tvlist[1].y < -z_test)
    vertex_ccodes[1] = CLIP_CODE_LY;
else
    vertex_ccodes[1] = CLIP_CODE_IY;
```

```
// Вершина 2.
```

```
z_test = z_factor*curr_poly->tvlist[2].z;
```

```
if (curr_poly->tvlist[2].y > z_test)
    vertex_ccodes[2] = CLIP_CODE_GY;
else
if (curr_poly->tvlist[2].x < -z_test)
    vertex_ccodes[2] = CLIP_CODE_LY;
else
    vertex_ccodes[2] = CLIP_CODE_IY;
```

```
// Проверяем, можно ли многоугольник исключить из
// рассмотрения. Для этого он должен полностью
// выходить за пределы верхней или нижней границ
// области обзора
```

```
if (((vertex_ccodes[0] == CLIP_CODE_GY) &&
    (vertex_ccodes[1] == CLIP_CODE_GY) &&
    (vertex_ccodes[2] == CLIP_CODE_GY) ) ||
```

```

    ((vertex_ccodes[0] == CLIP_CODE_LX) &&
     (vertex_ccodes[1] == CLIP_CODE_LX) &&
     (vertex_ccodes[2] == CLIP_CODE_LX) ) )

{
    // Помечаем многоугольник, полностью выходящий за
    // рамки области обзора
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // Переходим к следующему многоугольнику.
    continue;
} // if

} // if

// Отсечение и отбраковка по z-плоскостям
if (clip_flags & CLIP_POLY_Z_PLANE)
{
    // Отсечение и отбраковка на основе положения
    // многоугольника относительно плоскостей отсечения z.
    // Для каждой вершины определяем, находится ли она в
    // области отсечения или выходит за ее пределы. В
    // зависимости от этого присваиваем ей соответствующий
    // код отсечения, а затем производим отсечение всех
    // многоугольников, для которых это необходимо, по
    // ближней плоскости. В результате могут возникнуть
    // дополнительные многоугольники

    // Обнуляем счетчики вершин. Это поможет при
    // классификации конечных треугольников
    num_verts_in = 0;

    // Вершина 0,
    if (curr_poly->tvlist[0].z > cam->far_clip_z)
    {
        vertex_ccodes[0] = CLIP_CODE_GZ;
    }
    else
    if (curr_poly->tvlist[0].z < cam->near_clip_z)
    {
        vertex_ccodes[0] = CLIP_CODE_LZ;
    }
    else
    {
        vertex_ccodes[0] = CLIP_CODE_IZ;
        num_verts_in++;
    }

    // Вершина 1
    if (curr_poly->tvlist[1].z > cam->far_clip_z)
    {
        vertex_ccodes[1] = CLIP_CODE_GZ;
    }
}

```

```

else
if (curr_poly->tvlist[1].z < cam->near_clip_z)
{
    vertex_ccodes[1] = CLIP_CODE_LZ;
}
else
{
    vertex_ccodes[1] = CLIP_CODE_IZ;
    num_verts_in++;
}

// Вершина 2
if (curr_poly->tvlist[2].z > cam->far_clip_z)
{
    vertex_ccodes[2] = CLIP_CODE_GZ;
}
else
if (curr_poly->tvlist[2].z < cam->near_clip_z)
{
    vertex_ccodes[2] = CLIP_CODE_LZ;
}
else
{
    vertex_ccodes[2] = CLIP_CODE_IZ;
    num_verts_in++;
}

// Тривиальная проверка, выходят ли многоугольники
// полностью за пределы ближней или дальней
// плоскости z
if (((vertex_ccodes[0] == CLIP_CODE_GZ) &&
    (vertex_ccodes[1] == CLIP_CODE_GZ) &&
    (vertex_ccodes[2] == CLIP_CODE_GZ) ) ||

    ((vertex_ccodes[0] == CLIP_CODE_LZ) &&
    (vertex_ccodes[1] == CLIP_CODE_LZ) &&
    (vertex_ccodes[2] == CLIP_CODE_LZ) ) )

{
    // Помечаем многоугольник, полностью выходящий за
    // рамки области обзора
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // Переходим к следующему многоугольнику
    continue;
} // if

// Проверяем, лежит ли хоть одна вершина за пределами
// ближней плоскости отсечения
if (((vertex_ccodes[0] | vertex_ccodes[1] |
    vertex_ccodes[2]) & CLIP_CODE_LZ) )
{
    // Теперь все готово для отсечения многоугольников по

```

```

// ближайшей границе области обзора. Нет необходимости
// выполнять отсечение по дальней границе, поскольку ее
// пересечение не вызовет проблем. Возможны два случая:
// 1) одна вершина треугольника находится по одну
// сторону с областью обзора, а две - по разные
// стороны с этой областью; 2) две вершины находятся в
// области обзора, а одна - вне ее

// Этап 1: определение количества вершин, находящихся
// в положительном и отрицательном полупространстве
// по отношению к ближайшей плоскости отсечения
// Случай 1: легкий :)
if (num_verts_in == 1)
{
    // Производится отсечение треугольника по ближайшей
    // плоскости. Процедура отсечения выполняется для
    // каждой стороны, выходящей из внутренней вершины.
    // Для этого нужно найти их точки пересечения с
    // ближайшей плоскостью z. Это делается с помощью
    // параметрического уравнения прямой. Когда точка
    // пересечения найдена, параметры старой вершины
    // перезаписываются новыми. При необходимости заново
    // вычисляются координаты текстуры. В этом случае
    // новые данные внести несложно, поскольку усеченный
    // треугольник остается треугольником и его не нужно
    // разбивать заново. Ниже представлен случай 2, в
    // котором в результате усечения возникает два
    // треугольника. По крайней мере, один из них нужно
    // добавить в конец списка визуализации.

    // Этап 1: находим индекс, соответствующий
    // внутренней вершине
    if (vertex_ccodes[0] == CLIP_CODE_IZ)
    { v0 = 0; v1 = 1; v2 = 2; }
    else
    if (vertex_ccodes[1] == CLIP_CODE_IZ)
    { v0 = 1; v1 = 2; v2 = 0; }
    else
    { v0 = 2; v1 = 0; v2 = 1; }

    // Этап 2: отсекаем каждую сторону. Для этого
    // создается параметрическое уравнение прямой
    //  $p = v_0 + v_1 * t$ , из которого затем определяется, при
    // каком t координата z точки на прямой равна
    // near_clip_z. Полученное значение t подставляется в
    // два других уравнения, и определяются координаты x и
    // y точки пересечения. Все это можно было бы
    // выполнить с помощью кода высокого уровня, но для
    // экономии времени сделаем это вручную

    // Отсечение стороны v0->v1
    VECTOR4D_Build(&curr_poly->tvlist[v0].v,
        &curr_poly->tvlist[v1].v, &v);

```

```

// Пересечение происходит, когда z = near_clip_z;
// при этом t =
t1 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) /
        v.z);

// Теперь t подставляем в два других уравнения и
// находим координаты x,y точки пересечения стороны
// с плоскостью
xi = curr_poly->tvlist[v0].x + v.x * t1;
yi = curr_poly->tvlist[v0].y + v.y * t1;

// Перезаписываем параметры вершины новыми
curr_poly->tvlist[v1].x = xi;
curr_poly->tvlist[v1].y = yi;
curr_poly->tvlist[v1].z = cam->near_clip_z;

// Отсекаем сторону v0->v2
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
               &curr_poly->tvlist[v2].v, &v);

// Пересечение происходит, когда z = near_clip_z;
// при этом t =
t2 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) /
        v.z);

// Теперь подставляем t в два других уравнения и
// находим координаты x,y точки пересечения стороны
// с плоскостью
xi = curr_poly->tvlist[v0].x + v.x * t2;
yi = curr_poly->tvlist[v0].y + v.y * t2;

// Перезаписываем параметры вершины новыми.
curr_poly->tvlist[v2].x = xi;
curr_poly->tvlist[v2].y = yi;
curr_poly->tvlist[v2].z = cam->near_clip_z;

// Теперь известны оба параметра пересечения t1, t2.
// Проверяем, обладает ли треугольник текстурой. Если
// это так, отсекаем координаты текстуры
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
    ui = curr_poly->tvlist[v0].u0 +
        (curr_poly->tvlist[v1].u0 -
         curr_poly->tvlist[v0].u0) * t1;
    vi = curr_poly->tvlist[v0].v0 +
        (curr_poly->tvlist[v1].v0 -
         curr_poly->tvlist[v0].v0) * t1;
    curr_poly->tvlist[v1].u0 = ui;
    curr_poly->tvlist[v1].v0 = vi;

    ui = curr_poly->tvlist[v0].u0 +
        (curr_poly->tvlist[v2].u0 -

```

```

    curr_poly->tvlist[v0].u0)*t2;
vi = curr_poly->tvlist[v0].v0 +
    (curr_poly->tvlist[v2].v0 -
    curr_poly->tvlist[v0].v0)*t2;
curr_poly->tvlist[v2].u0 = ui;
curr_poly->tvlist[v2].v0 = vi;
} // if textured

// Наконец, удаляем предварительно вычисленную
// длину нормали. Ее придется вычислять заново!!!

// Строим векторы u и v
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v2].v, &v);

// Вычисляем векторное произведение.
VECTOR4D_Cross(&u, &v, &n);

// Быстрое вычисление длины нормали и сохранение ее
// в соответствующей переменной
curr_poly->nlength = VECTOR4D_Length_Fast(&n);
} // if
else
if (num_verts_in == 2)
{ // num_verts = 2

// Остается случай, когда num_verts_in = 2.
// Треугольник нужно отсечь по ближней плоскости.
// Процедура отсечения выполняется для каждой
// стороны, выходящей из внутренней вершины. Для
// этого нужно найти их точки пересечения с ближней
// плоскостью z. Это делается с помощью
// параметрического уравнения прямой. Однако, в
// отличие от рассмотренного ранее случая 1 усеченный
// многоугольник будет разбит на два треугольника.
// Поэтому в процессе отсечения результаты
// сохраняются в новом треугольнике, который
// помещается в конец списка визуализации. В конце
// информация о старом многоугольнике
// перезаписывается новыми данными

// Этап 0: копируем информации о многоугольнике
memcpy(&temp_poly, curr_poly, sizeof(POLYF4DV2));

// Этап 1: находим индекс внешней вершины
if (vertex_ccodes[0] == CLIP_CODE_LZ)
{ v0 = 0; v1 = 1; v2 = 2; }
else
if (vertex_ccodes[1] == CLIP_CODE_LZ)
{ v0 = 1; v1 = 2; v2 = 0; }
else

```

```
{ v0 = 2; v1 = 0; v2 = 1; }
```

```
// Этап 2: отсечение каждой стороны. Для этого
// создается параметрическое уравнение прямой
//  $p \cdot v0 + v01 \cdot t$ , из которого затем определяется,
// при каком  $t$  координата  $z$  точки на прямой равна
// near_clip_z. Полученное значение  $t$  подставляется
// в два других уравнения, и определяются координаты
//  $x$  и  $y$  точки пересечения. Все это можно было бы
// выполнить с помощью кода высокого уровня, но для
// экономии времени сделаем это вручную
```

```
// Отсечение стороны  $v0 \rightarrow v1$ .
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v1].v, &v);
```

```
// Пересечение происходит, когда  $z = \text{near\_clip\_z}$ ;
// при этом  $t =$ 

$$t1 = \frac{(\text{cam} \rightarrow \text{near\_clip\_z} - \text{curr\_poly} \rightarrow \text{tvlist}[v0].z)}{v.z};$$

```

```
// Теперь  $t$  подставляем в два других уравнения и
// находим координаты  $x, y$  точки пересечения стороны
// с плоскостью

$$x01i = \text{curr\_poly} \rightarrow \text{tvlist}[v0].x + v.x * t1;$$


$$y01i = \text{curr\_poly} \rightarrow \text{tvlist}[v0].y + v.y * t1;$$

```

```
// Отсечение стороны  $v0 \rightarrow v2$ .
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v2].v, &v);
```

```
// Пересечение происходит, когда  $z = \text{near\_clip\_z}$ ;
// при этом  $t =$ 

$$t2 = \frac{(\text{cam} \rightarrow \text{near\_clip\_z} - \text{curr\_poly} \rightarrow \text{tvlist}[v0].z)}{v.z};$$

```

```
// Теперь  $t$  подставляем в два других уравнения и
// находим координаты  $x, y$  точки пересечения стороны
// с плоскостью

$$x02i = \text{curr\_poly} \rightarrow \text{tvlist}[v0].x + v.x * t2;$$


$$y02i = \text{curr\_poly} \rightarrow \text{tvlist}[v0].y + v.y * t2;$$

```

```
// Теперь найдены обе точки пересечения. Необходимо
// переписать замещающую вершину многоугольника  $O$ 
// одной из точек пересечения. При этом образуется
// один из двух треугольников, которые получаются
// в результате разбиения
```

```
// Перезаписываем информацию о вершине

$$\text{curr\_poly} \rightarrow \text{tvlist}[v0].x = x01i;$$


$$\text{curr\_poly} \rightarrow \text{tvlist}[v0].y = y01i;$$


$$\text{curr\_poly} \rightarrow \text{tvlist}[v0].z = \text{cam} \rightarrow \text{near\_clip\_z};$$

```

```

// Теперь начинается трудная часть. С помощью обеих
// точек пересечения и вершины v2 необходимо
// аккуратно образовать новый треугольник. Он будет
// помещен в конец списка визуализации. Однако пока
// что этот треугольник заносится во временную
// переменную temp_poly

// Вершину v2 оставим без изменений, вершину v1
// перезапишем данными, соответствующими точке v01,
// а вершину v0 - данными, соответствующими
// точке v02
temp_poly.tvlist[v1].x = x01i;
temp_poly.tvlist[v1].y = y01i;
temp_poly.tvlist[v1].z = cam->near_clip_z;

temp_poly.tvlist[v0].x = x02i;
temp_poly.tvlist[v0].y = y02i;
temp_poly.tvlist[v0].z = cam->near_clip_z;

// Теперь известны оба параметра пересечения t1, t2.
// Проверяем, обладает ли треугольник текстурой. Если
// это так, отсекаем координаты текстуры
if (curr_poly->attr &
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
    // Определяем новые координаты текстуры для
    // полученного при отсечении треугольника 1.
    u01i = curr_poly->tvlist[v0].u0 +
        (curr_poly->tvlist[v1].u0 -
         curr_poly->tvlist[v0].u0) * t1;
    v01i = curr_poly->tvlist[v0].v0 +
        (curr_poly->tvlist[v1].v0 -
         curr_poly->tvlist[v0].v0) * t1;

    // Определяем новые координаты текстуры для
    // полученного при отсечении треугольника 2
    u02i = curr_poly->tvlist[v0].u0 +
        (curr_poly->tvlist[v2].u0 -
         curr_poly->tvlist[v0].u0) * t2;
    v02i = curr_poly->tvlist[v0].v0 +
        (curr_poly->tvlist[v2].v0 -
         curr_poly->tvlist[v0].v0) * t2;

    // Записываем все данные одновременно
    // Многоугольник 1
    curr_poly->tvlist[v0].u0 = u01i;
    curr_poly->tvlist[v0].v0 = v01i;

    // Многоугольник 2
    temp_poly.tvlist[v0].u0 = u02i;
    temp_poly.tvlist[v0].v0 = v02i;
    temp_poly.tvlist[v1].u0 = u01i;
    temp_poly.tvlist[v1].v0 = v01i;
}

```

```

} // if textured

// Наконец, удаляем предварительно вычисленную
// длину нормали. Ее придется вычислять заново!!!

// Сначала многоугольник 1; он остается на месте

// Строим векторы и и v
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
    &curr_poly->tvlist[v2].v, &v);

// Вычисляем векторное произведение
VECTOR4D_Cross(&u, &v, &n);

// Быстрое вычисление длины нормали и сохранение ее
// в соответствующей переменной.
curr_poly->nlength = VECTOR4D_Length_Fast(&n);

// Теперь многоугольник 2, temp_poly
// Построение векторов и и v
VECTOR4D_Build(&temp_poly.tvlist[v0].v,
    &temp_poly.tvlist[v1].v, &u);
VECTOR4D_Build(&temp_poly.tvlist[v0].v,
    &temp_poly.tvlist[v2].v, &v);

// Вычисление векторного произведения
VECTOR4D_Cross(&u, &v, &n);

// Быстрое вычисление длины нормали и сохранение ее
// в соответствующей переменной.
temp_poly.nlength = VECTOR4D_Length_Fast(&n) ;

// Теперь все готово. Помещаем многоугольник в
// список. Если он не подходит, это не имеет
// значения; в этом случае функция возвращает 0
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list,
    &temp_poly);

} // else

} // if

} // if

} // for poly

} // Clip_Polys_RENDERLIST4DV2

```

Должен признаться, что сам я редко читаю приведенный в книгах код. Я просто не могу его понять, потому что это очень сложно. И хотя иногда я изучаю каждую строку кода и пытаюсь понять, что он **означает**, я все же предпочитаю абстрактные объяснения и псевдокод. Тем не менее, некоторые учатся по-другому, вот для **них** я и привожу в своих книгах листинги. Я пытаюсь сказать, что даже если вы относитесь к коду так же, как и я, пожалуйста, попытайтесь все же изучить приведенную функцию строка за строкой и понять ее. Она может стать **основой** многих **вещей**, и на самом деле она достаточно простая, потому что многие ее части повторяются.

Использовать эту функцию до смешного просто. Она вызывается **следующим** образом,

```
Clip_Polys_RENDERLIST4DV2(&rend_list,&cam,
    CLIP_POLY_Z_PLANE |
    CLIP_POLY_X_PLANE |
    CLIP_POLY_Y_PLANE);
```

В функцию передаются список визуализации, параметры камеры и управляющие флаги, а затем функция делает все остальное. Обратите внимание, что независимо можно задавать отсечение по плоскостям в направлении осей x, y и z. Я предпочитаю **устанавливать** все возможные виды отсечения, чтобы отбраковать как можно больше многоугольников!

## Демонстрационная программа

В качестве примера использования функции отсечения и общих принципов была создана еще одна демонстрационная программа, моделирующая вращение в пространстве некоторого загадочного объекта. Программа содержится в файле с именем **DEMO110\_1.CPP|EXE** на прилагаемом компакт-диске. Чтобы скомпилировать эту программу, как обычно, понадобятся файлы **T3DLIB1..8.CPP|H** и библиотечные файлы DirectX. Копия экрана, полученная в результате работы программы, показана на рис. 10.23. В процессе работы объект загружается и отображается на экране, медленно вращаясь. Его можно приблизить к точке наблюдения или удалить от нее, а также сместить или даже полностью вывести за пределы экрана.

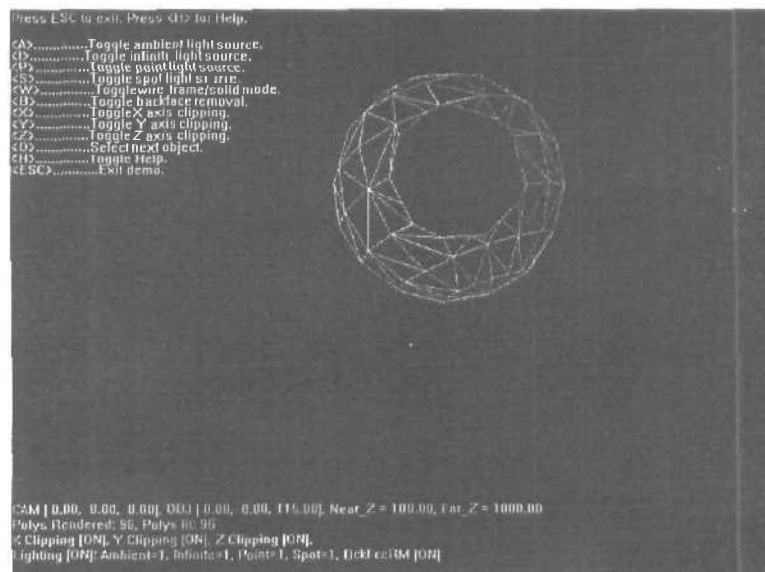


Рис. 10.23. Копия экрана демонстрационной программы отсечения

У этой демонстрационной программы двойное предназначение. Во-первых, она позволяет понять, что если отсечение выполняется на уровне объекта, невозможно отбраковывать или отсекалть отдельно взятые многоугольники. Таким образом, даже если объект частично находится в области обзора, он в любом случае освещается и преобразовывается как единое целое. В данном случае отсечение и отбраковка производится на уровне многоугольников. Следите за статистической информацией, которая отображается в нижней части экрана. Список управляющих **клавиш** представлен в справке, которая тоже отображается на экране, **но**, как обычно, объект перемещается с помощью клавиш управления курсором, смена объекта производится с помощью клавиши <O>, а для включения или отключения отсечения по плоскостям **x**, и **z** служат клавиши <X>, <Y> и <Z>, соответственно.

### Обновление системы освещения

Перед тем как продолжить, необходимо рассмотреть небольшую деталь, имеющую отношение к системе освещения. До сих пор освещение производилось в мировой системе координат, поскольку именно в этой системе задавалось положение источников освещения и геометрических элементов. Однако теперь такое состояние дел оборачивается огромными затратами вычислительных ресурсов. Освещение можно отложить до тех пор, пока не будет выполнено отсечение, и сэкономить на освещении отбракованных и отсеченных многоугольников. Для этого координаты и направления источников освещения нужно **преобразовать** в систему координат камеры (потому что именно в эту систему координат будет осуществлен переход после этапа отсечения), и только затем смоделировать освещение. Это можно сделать несколькими способами. Можно было бы переписать функции освещения и преобразовать параметры положения и ориентации источников, исходя из положения камеры (поскольку оно является **параметром**). Однако здесь возникает проблема, которая состоит в том, что преобразование источников освещения "на лету" приводит к тому, что если функции освещения вызываются на уровне объекта, эти преобразования необходимо выполнять снова и снова (а для этого вызывать функцию). Лучше преобразовать сами источники в систему координат камеры.

Единственное, что нужно учесть, — если положение и ориентация источников **освещения** преобразуется в систему координат камеры, то где-то должны храниться старые параметры источников освещения. К сожалению, нам приходится вернуться к структурам, в которых хранятся параметры источников, и обновить их. Вместо структуры `LIGHTV1` будет создана структура `LIGHTV2` (версия 2.0), в которую добавлено место для хранения новых данных, **описывающих** положение и ориентацию источников освещения после перехода в систему координат камеры. Кроме этого, понадобится переписать все функции, работающие с источниками освещения, поскольку теперь в них вместо структуры `LIGHTV1` будет использоваться структура `LIGHTV2`. К счастью, сами функции почти не придется изменять; достаточно заменить строку `LIGHTV1` строкой `LIGHTV2`. Ознакомимся с новой структурой `LIGHTV2`.

```
// Структура, в которой хранятся параметры
// источника освещения; версия 2.0.
typedef struct LIGHTV2_TYP
{
    int state;    // Состояние источника
    int id;       // Идентификатор источника
    int attr;     // Тип источника и его
                // дополнительные спецификаторы

    RGBAV1 c_ambient; // Интенсивность общего освещения
```

```

RGBA1 c_diffuse; // Интенсивность диффузного освещения
RGBA1 c_specular; // Интенсивность зеркального освещения

POINT4D pos; // Положение источника в мировых координатах
POINT4D tpos; // и в системе координат камеры
VECTOR4D dir; // Направление источника в мировой системе
VECTOR4D tdir; // координат и в системе координат камеры
float kc, kl, kq; // Коэффициенты затухания
float spot_inner; // Внутренний угол светового пятна
float spot_outer; // Внешний угол светового пятна
float pf; // Показатель степени, определяющий
// затухание светового пятна

intiaux1,iaux2; // Вспомогательные переменные .
floataux1,faux2;
void *ptr;

} LIGHTV2, *LIGHTV2_PTR;

```

Новые поля, в которых будут храниться параметры, определяющие новое положение и ориентацию источника после перехода в систему координат камеры, выделены полужирным шрифтом. Таким образом, для локального положения и ориентации источника по-прежнему служат переменные-члены `pos` и `dir`, а единственное отличие новой версии структуры от старой заключается в том, что соответствующие данные в системе отсчета камеры заносятся в переменные `tpos` и `tdir`. Вопрос в том, как осуществить переход из одной системы в другую? Для этого понадобится функция, код которой представлен ниже.

```

void Transform_LIGHTSV2(
    LIGHTV2_PTR lights, // Массив с преобразуемыми
                        // источниками
    int num_lights, // Количество преобразуемых
                  // источников
    MATRIX4X4_PTR mt, // Матрица перехода
    int coord_select) // Выбор системы координат
{
    // Эта функция выполняет преобразование всех источников
    // с помощью заданной матрицы перехода. Например, она
    // может использоваться для перехода в систему координат
    // камеры. Это позволяет правильно моделировать
    // освещение, если функция освещения вызывается ПОСЛЕ
    // того, как многоугольники также преобразованы в
    // систему координат камеры. Возможно, позже эта функция
    // будет оптимизирована: в ней будет задаваться тип
    // преобразуемого источника, чтобы изменение направления
    // выполнялось только для тех источников, для которых
    // это необходимо. Но количество вершин может составить
    // несколько тысяч, поэтому оптимизация, касающаяся
    // десятка источников, мало повлияет на
    // производительность программы

    // ПРИМЕЧАНИЕ. Эту функцию НЕОБХОДИМО вызывать даже в
    // тех случаях, если параметры источников не нуждаются в
    // преобразовании и освещение выполняется в мировых
    // координатах. В этом случае локальные положения и

```

```

// ориентации источников также НЕОБХОДИМО скопировать в
// рабочие переменные, чтобы в процессоре освещения
// использовались переменные pos->tpos, dir->tdir. Если
// переход в систему координат не происходит, функция
// вызывается с параметром
// TRANSFORM_COPY_LOCAL_TO_TRANS, а для матрицы задается
// значение параметра NULL

int curr_light;    // Текущий источник в цикле
MATRIX4X4 mr;     // Используется при построении матрицы
                  // поворота

// Нужно произвести поворот векторов, задающих
// направление источников. Однако при этом параметры
// сдвига в матрице необходимо обнулить, иначе получим
// неверные результаты. Итак, копируем матрицу и
// обнуляем в ней параметры сдвига

if (mt!=NULL)
{
    MAT_COPY_4X4(mt, &mr);
    // Обнуление параметров сдвига
    mr.M30 - mr.M31 = mr.M32 - 0;
} // if

// Какие координаты следует преобразовывать?
switch(coord_select)
{
    case TRANSFORM_COPY_LOCAL_TO_TRANS:
    {
        // Цикл по всем источникам
        for (curr_light=0;
            curr_light < num_lights;
            curr_light++)
        {
            lights[curr_light].tpos -
                lights[curr_light].pos;
            lights[curr_light].tdir =
                lights[curr_light].dir;
        } // for

        } break;

    case TRANSFORM_LOCAL_ONLY:
    {
        // Цикл по всем источникам
        for (curr_light = 0;
            curr_light < num_lights;
            curr_light++)
        {
            // Переход из локальной или мировой
            // системы координат
            POINT4D presult; // Сохранение
                            // результатов каждого преобразования

            // Преобразование положения каждого

```

```

        // источника
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].pos, mt,
            &presult);

        // Сохранение результатов
        VECTOR4D_COPY(&lights[curr_light].pos,
            &presult);

        // Преобразование вектора направления
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].dir, &mr,
            &presult);

        // Сохранение результатов
        VECTOR4D_COPY(&lights[curr_light].dir,
            &presult);
    } // for

} break;

case TRANSFORM_TRANS_ONLY:
{
    // Цикл по всем источникам
    for (curr_light= 0;
        curr_light < num_lights;
        curr_light++)
    {
        // Преобразование каждого
        // "преобразованного" источника
        POINT4D presuit; // Сохранение
        // результатов каждого преобразования

        // Преобразование положения каждого
        // источника
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].tpos, mt,
            &presult);

        // Сохранение результатов
        VECTOR4D_COPY(&lights[curr_light].tpos,
            &presult);

        // Преобразование вектора направления
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].tdir, &mr,
            &presult);

        // Сохранение результатов
        VECTOR4D_COPY(&lights[curr_light].tdir,
            &presult);
    } // for

} break;

case TRANSFORM_LOCAL_TO_TRANS:

```

```

{
    // Цикл по всем источникам
    for (curr_light = 0;
        curr_light < num_lights;
        curr_light++)
    {
        // Преобразование каждого источника,
        // заданного в локальной или мировой
        // системе координат, и помещение
        // результатов в соответствующие
        // переменные. Это обычный способ вызова
        // функции
        POINT4D presult; // Сохранение
        // результатов каждого преобразования

        // Преобразование положения каждого
        // источника
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].pos, mt,
            &lights[curr_light].tpos);

        // Преобразование вектора направления
        Mat_Mul_VECTOR4D_4X4(
            &lights[curr_light].dir, &mr,
            &lights[curr_light].tdir);
    } // for

    } break;

    default: break;

} // switch
} // Transform_LIGHTSV2

```

При вызове этой функции достаточно задать параметры источников освещения, их количество, матрицу преобразования и, наконец, систему координат. Эта функция похожа на другие функции преобразования, которые нам уже встречались, и работает точно так же. В 99 случаях из 100 ее вызов будет выглядеть примерно следующим образом.

```

Transform_LIGHTSV2(lights2, 4, &cam.mcam,
    TRANSFORM_LOCAL_TO_TRANS);

```

В этом случае параметры четырех источников освещения, содержащиеся в массиве `lights2[]`, преобразуются с помощью матрицы `cam.mcam`. Результаты преобразования записываются в специально предназначенные для этого переменные. Теперь, когда у нас имеются обновленные источники, в файле `T3DLIB8.CPP` для них объявлен новый массив.

```

LIGHTV2 lights2[MAX_LIGHTS]; // Источники освещения системы

```

Кроме того, в заголовочном файле для источников необходимо определить новые директивы `#define`.

```

// Директивы typedef для типов источников; версия 2.
#define LIGHTV2_ATTR_AMBIENT 0x0001
    // Источник общего освещения
#define LIGHTV2_ATTR_INFINITE 0x0002
    // Бесконечно удаленный источник

```

```

fdefine LIGHTV2_ATTR_DIRECTIONAL 0x0002
    // Бесконечно удаленный источник (псевдоним)
tfdefine LIGHTV2_ATTR_POINT 0x0004
    // Точечный источник
tfdefine LIGHTV2_ATTR_SPOTLIGHT1 0x0008
    // Световое пятно вида 1 (простая модель).
tfdefine LIGHTV2_ATTR_SPOTLIGHT2 0x0010
    // Световое пятно вида 2 (сложная модель)

#define LIGHTV2_STATE_ON 1 // Включенный
#define LIGHTV2_STATE_OFF 0 // Выключенный

```

```

// Флаги, управляющие преобразованием.
#define TRANSFORM_COPY_LOCAL_TO_TRANS 3
    // Копирование данных без изменений

```

Приведенные выше директивы имеют то же предназначение, что и директивы, использовавшиеся в структуре `LIGHTV1`, только имена переменных теперь начинаются префиксом `LIGHTV2_`. Кроме того, специально для функции `Transform_LIGHTSV2()` определена одна дополнительная команда `TRANSFORM_COPY_LOCAL_TO_TRANS`. Она используется в тех ситуациях, когда преобразование выполнять не нужно, и параметры источников в локальной системе координат просто копируются в соответствующие переменные, предназначенные для преобразованных параметров.

Наконец, необходимо переписать функции **освещения**. В основном модификации будут минимальными, сводящимися к замене имен. В процессе вычислений вместо переменных `pos` и `dir`, предназначенных для хранения положения и направления источников, необходимо использовать переменные `tpos` и `tdir`. Листинги всех функций приводиться не будут, поскольку это заняло бы слишком много места, однако ниже представлены их прототипы, заданные в заголовочном файле.

```

int Light_OBJECT4DV2_World2_16(
    OBJECT4DV2_PTR obj,    // Обрабатываемый объект
    CAM4DV1_PTR cam,       // Положение камеры
    LIGHTV2_PTR lights,    // Список источников (их
                          // может быть несколько)
    int max_lights);       // Максимальное количество
                          // источников в списке

int Light_OBJECT4DV2_World2(
    OBJECT4DV2_PTR obj,    // Обрабатываемый объект
    CAM4DV1_PTR cam,       // Положение камеры
    LIGHTV2_PTR lights,    // Список источников (их
                          // может быть несколько)
    int max_lights);       // Максимальное количество
                          // источников в списке

int Light_RENDERLIST4DV2_World2(
    RENDERLIST4DV2_PTR rend_list, // Обрабатываемый
                                // список
    CAM4DV1_PTR cam,           // Положение камеры
    LIGHTV2_PTR lights,        // Список источников (их
                                // может быть несколько)
    int max_lights);          // Максимальное количество
                                // источников в списке

```

```

int Light_RENDERLIST4DV2_World2_16(
    RENDERLIST4DV2_PTR rend_list, // Обрабатываемый
                                   // список
    CAM4DV1_PTR cam,              // Положение камеры
    LIGHTV2_PTR Lights,           // Список источников (их
                                   // может быть несколько)
    int max_lights);              // Максимальное количество
                                   // источников в списке

// Система освещения.
int Init_Light_LIGHTV2(
    LIGHTV2_PTR lights,           // Обрабатываемый массив с
                                   // источниками (новая переменная)
    int index,                    // Индекс создаваемого источника
                                   // (0..MAX_LIGHTS-1)
    int _state,                   // Состояние источника
    int _attr,                    // Вид источника и его
                                   // дополнительные спецификаторы
    RGBAV1 _c_ambient,            // Интенсивность общего освещения
    RGBAV1 _c_diffuse,            // Интенсивность рассеянного света
    RGBAV1 _c_specular,           // Интенсивность зеркального
                                   // освещения
    POINT4D_PTR _pos,             // Положение источника
    VECTOR4D_PTR _dir,            // Направление источника
    float _kc,                    // Коэффициенты затухания
    float _kl,
    float _kq,
    float _spot_inner,            // Внутренний угол светового
                                   // пятна
    float _spot_outer,            // Внешний угол светового пятна
    float _pf);                  // Показатель степени,
                                   // определяющий затухание
                                   // светового пятна

int Reset_Lights_LIGHTV2(
    LIGHTV2_PTR Lights,           // Обрабатываемый массив с пара-
                                   // метрами источников освещения
    int max_lights);              // (новая переменная)
                                   // Количество источников в системе

```

Единственное отличие этих функций от их **предыдущих** версий заключается в наличии одного дополнительного параметра: в качестве первого параметра в них задается указатель на массив, в **котором** хранятся характеристики источников. Это позволяет обрабатывать в осветительной системе более одного набора глобальных источников.

#### ВНИМАНИЕ

Задавая положение и направление источников в структурах **POINT4D** и **VECTOR4D**, не забывайте, что последний параметр всегда должен быть равен 1.0. Если его не задать, то в процессе **преобразования** параметров источника компоненты матрицы, соответствующие сдвигу, не окажут никакого воздействия!

Ознакомьтесь с демонстрационными программами, **иллюстрирующими** вызовы новых функций. Основная причина, по которой нам понадобилось переделать функции освещения, заключается в том, что **освещение** нужно выполнять в системе координат камеры. Моделирование освещения откладывается до тех пор, пока мы не перейдем в эту систему координат, чтобы пришлось освещать минимальное количество многоугольников.

## Игры с ландшафтами

Причины, по которым выполняется отсечение, могут быть разными. Однако основной мотив — минимизировать количество многоугольников, которые будут направлены для обработки в дальнейших стадиях игрового конвейера, и предотвратить проецирование многоугольников с отрицательными или нулевыми координатами  $z$ , что может вызвать проблемы в аксонометрических вычислениях. До настоящего времени все демонстрационные программы были разработаны для объектов со сравнительно небольшим количеством многоугольников, поэтому в большинстве случаев вероятность наличия многоугольников с отрицательными или нулевыми координатами  $z$  была пренебрежимо мала. Однако у нас все еще нет возможности загружать модели игрового уровня или интерьера помещений, состоящие из большого количества многоугольников. Я решил разработать черновой вариант функции, генерирующей очень большие ландшафты (произвольного размера), и представляющей их в виде мозаики из треугольников, цвет которых зависит от высоты расположения.

Здесь возникают две проблемы. Во-первых, у нас нет никаких элементов, предназначенных для хранения трехмерных объектов, кроме структуры `OBJECT4DV2`, а она не приспособлена для хранения таких больших каркасов. Однако пока что можно использовать и ее. Из-за чего действительно возникает проблема, так это из-за того, что ландшафты имеют печальную славу объектов, число составляющих многоугольников которых находится в пределах от нескольких тысяч до нескольких миллионов. Однако, призвав на помощь здравый смысл, 99% многоугольников можно отбраковать, воспользовавшись алгоритмами разбиения пространства на секторы. В этих алгоритмах ландшафты огромных размеров создаются путем заполнения пространства фрагментами ландшафта; при этом обрабатываются лишь те фрагменты, которые находятся в области обзора. По сути, каждый такой фрагмент сам выступает в роли объекта, и у нас образуется большой массив, состоящий из фрагментов ландшафта (рис. 10.24).

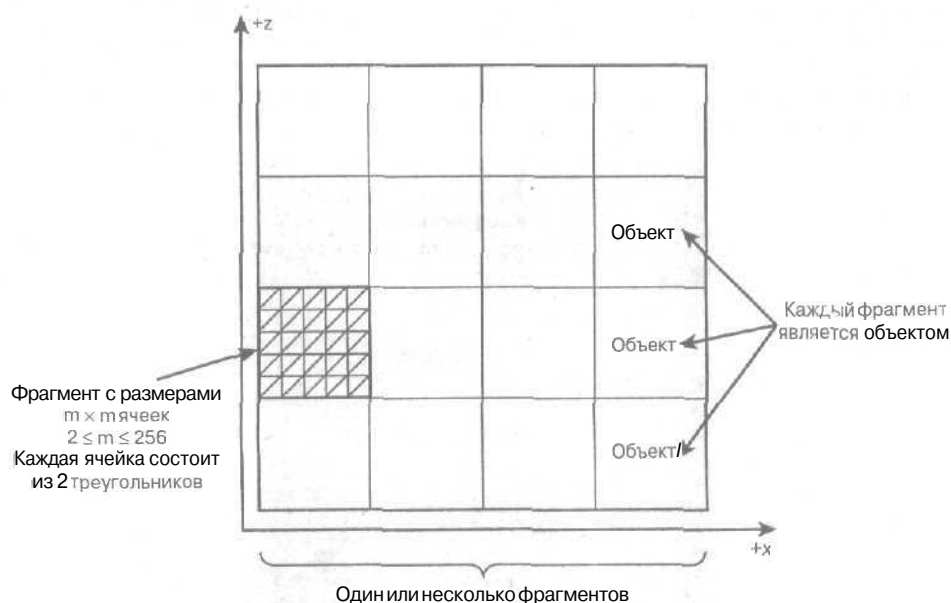


Рис. 10.24. Пример создания большого ландшафта путем разбиения его на фрагменты, которые являются сложными объектами

Можно воспользоваться несколькими объектами и составить из них мозаику, в результате чего образуется ландшафт огромных размеров. Затем с помощью отбраковки из рассмотрения будут исключены фрагменты ландшафта, не входящие в область обзора. Но здесь возникает вторая, более коварная проблема. Она заключается в том, что даже если обрабатывается один фрагмент ландшафта, то в нем может использоваться  $32 \times 32$ ,  $64 \times 64$  или  $256 \times 256$  значений высоты, что составляет  $31 \times 31$ ,  $63 \times 63$  или  $255 \times 255$  ячеек (количество ячеек в каждом направлении на единицу меньше, чем количество точек, в которых задается высота). Это происходит из-за того, что высоты задаются в каждом углу ячейки. Таким образом, в сумме фрагмент содержит  $2 \cdot 31^2$ ,  $2 \cdot 63^2$  или  $2 \cdot 255^2$  многоугольников!

Даже с учетом отсечения и отбраковки мы не справимся с таким их количеством. На самом деле процессор ландшафтов на ходу производит анализ степени детализации, сочетая возможность исключения ненужных многоугольников и добавления их туда, где они нужны. Если в поле зрения находится один фрагмент ландшафта, состоящий, скажем, из  $64 \times 64$  ячеек, в каждой из которых содержится по два треугольника, то всего в состав такой модели входит **8192** треугольников. Однако после сокращения их количества, которое при этом происходит, их остается буквально несколько сотен! Я не собираюсь описывать здесь эти алгоритмы; в конце раздела приведены некоторые ссылки, так что вы можете продолжить изучение этого вопроса самостоятельно.

Здесь же я хочу описать создание простой функции, предназначенной для генерирования ландшафта. Достаточно вызвать ее один раз — и массив помещается в объект, с которым можно работать как обычно.

## Функция, генерирующая ландшафты

Ландшафты можно генерировать несколькими способами: с помощью случайных чисел, с помощью данных, полученных со спутника, с помощью цветных карт высот или путем сочетания этих методов. Мы собираемся воспользоваться картой высот. План действий состоит в том, чтобы загрузить **256-цветовое** битовое изображение, используемое для представления моделируемого высотного поля, а также цветовую текстуру, которая накладывается на ландшафт. Одно из таких полей представлено на рис. 10.25 (конечно же, в таком масштабе, чтобы иметь возможность разместить его на рисунке). Обратите внимание, как различным цветам сопоставляются разные высоты.

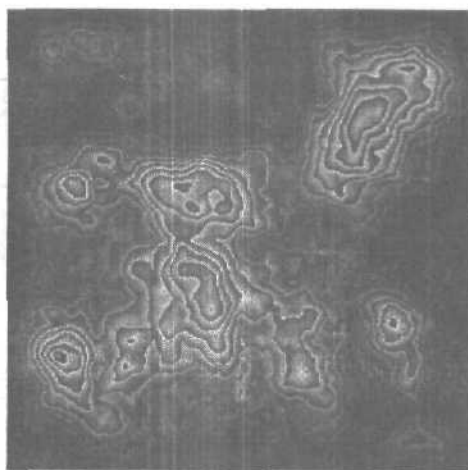
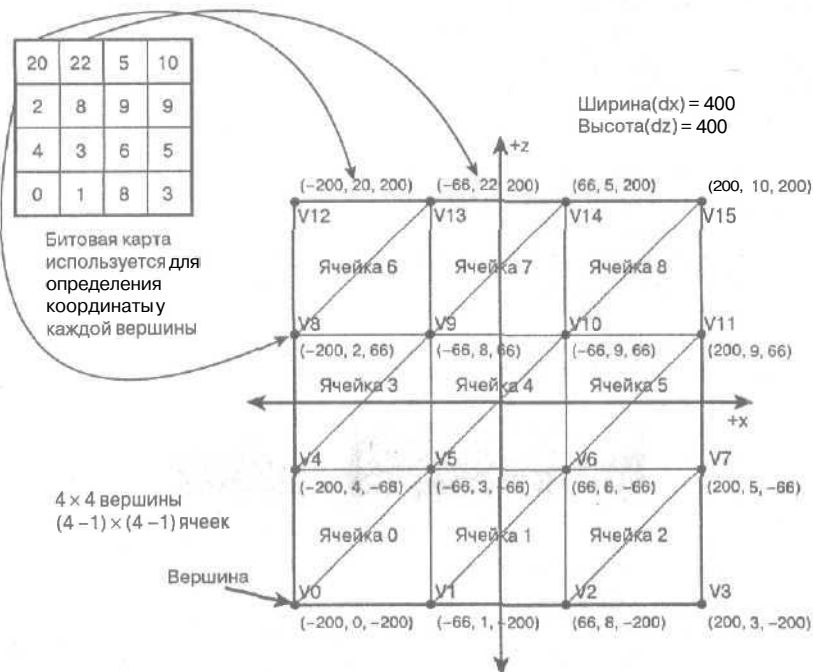


Рис. 10.25. Использование битового образа для представления высот ландшафта

Генерирование поля высот с помощью данных — это не что иное как несколько циклов *for*, при написании которых следует соблюдать крайнюю *осторожность*, чтобы не ошибиться, задавая их пределы. По ходу работы алгоритма для каждого квадрата сетки, высота каждой вершины которого определяется как индекс цвета, умноженный на множитель масштабирования, генерируются два треугольника. Кроме того, нужно знать размеры карты *высот*, т.е. ее ширину и длину. Суть в том, что в плоскости *xz* будет сгенерирована карта высот, а данные битового образа используются для определения высоты каждого четырехугольника, в вершинах которого задаются высоты. Затем каждый четырехугольник разбивается на два треугольника, которые помещаются в список визуализации объекта.



Как уже упоминалось, можно очень легко ошибиться, задав **отличающиеся** на единицу границы циклов. Если вы не понимаете, о чем идет речь, ответьте на такой простой вопрос: сколько чисел находится между 0 и 10? Оказывается, их 11! Сколько площадок для стоянки получится, если нарисовать пять разделительных линий? Их будет четыре. Вот это и есть примеры ситуаций, в которых легко ошибиться на единицу. Поэтому нужно убедиться в том, что все подсчитано правильно. В частности, это очень важно в **случае**

ях, подобных искусственному созданию каркасов, поскольку в результате ошибки на единицу может произойти обращение за пределы выделенной памяти и т.п. неприятности. Мы собираемся буквально шаг за шагом пройти алгоритм для карты высот очень маленьких размеров. Для этого в подобных случаях я обычно использую **битовый** образ размером 1x1, а затем — 4x4. Такой способ позволяет убедиться, что алгоритм работает для минимальных размеров, а затем по **индукции** можно предположить, что он работает и в произвольном случае.

Процесс искусственного построения структуры **OBJECT4DV2** выглядит не так уж страшно. Суть в том, что **информация**, которая обычно считывается с диска, просто генерируется и **направляется** в объект. Нам понадобится список вершин, список многоугольников, их количество, а также атрибуты каждого многоугольника. Все просто за исключением атрибутов многоугольников, в которых задается **информация**, касающаяся освещения и текстуры. Эти данные необходимо вручную передать в функцию с **помощью** следующих параметров.

```
// Директивы #defines для многоугольников и поверхностей;  
// версия 2
```

```
// Атрибуты многоугольников и их поверхностей.  
tfdefine POLY4DV2_ATTR_2SIDED      0x0001  
#define POLY4DV2_ATTR_TRANSPARENT  0x0002  
#define POLY4DV2_ATTR_8BITCOLOR    0x0004  
#define POLY4DV2_ATTR_RGB16        0x0008  
#define POLY4DV2_ATTR_RGB24        0x0010  
  
#define POLY4DV2_ATTR_SHADE_MODE_PURE 0x0020  
tfdefine POLY4DV2_ATTR_SHADE_MODE_CONSTANT 0x0020  
// (Псевдоним)  
#define POLY4DV2_ATTR_SHADE_MODE_EMISSIVE 0x0020  
// (Псевдоним)  
  
#define POLY4DV2_ATTR_SHADE_MODE_FLAT 0x0040  
#define POLY4DV2_ATTR_SHADE_MODE_GOURAUD 0x0080  
#define POLY4DV2_ATTR_SHADE_MODE_PHONG 0x0100  
#define POLY4DV2_ATTR_SHADE_MODE_FASTPHONG 0x0100  
// (Псевдоним)  
#define POLY4DV2_ATTR_SHADE_MODE_TEXTURE 0x0200  
  
// Новая часть  
#define POLY4DV2_ATTR_ENABLE_MATERIAL 0x0800  
// При освещении используется материал  
tfdefine POLY4DV2_ATTR_DISABLE_MATERIAL 0x1000  
// При освещении используется только  
// базовый цвет (эмуляция версии 1.0)
```

Полужирным шрифтом выделены атрибуты, с которыми мы имеем дело. Поговорим немного о них. Функция, генерирующая ландшафт, может работать в 8-битовом или 16-битовом режимах, поэтому в директивах **#define** необходимо определить флаг, управляющий режимом.

```
POLY4DV2_ATTR_8BITCOLOR  
POLY4DV2_ATTR_RGB16
```

Далее, функция будет поддерживать постоянное затенение, плоское затенение и затенение по Гуро, поскольку с геометрической точки зрения различия при этом сводятся к тому, нужно ли вычислять нормали. Однако это не должно нас беспокоить, потому что как только каркас будет готов, можно вызвать следующие функции.

```
// Вычисление длин нормалей к многоугольникам  
Compute _OBJECT4DV2_Poly_Normals(obj);
```

```
// Вычисление нормалей в вершинах для  
// многоугольников с затенением по Гуро  
Compute _OBJECT4DV2_Vertex_Normals(obj);
```

В процессе их работы для многоугольников с затенением по Гуро будут вычислены нормали в вершинах, а для процедуры удаления обратных поверхностей и освещения многоугольников — обычные нормали.

Режим затенения задается с помощью одной из следующих констант.

```
POLY4DV2_ATTR_SHADE_MODE_EMISSIVE  
POLY4DV2_ATTR_SHADE_MODE_FLAT  
POLY4DV2_ATTR_SHADE_MODE_GOURAUD
```

Далее, если мы хотим на ландшафт наложить текстуру, это делается с помощью следующей константы.

```
POLY4DV2_ATTR_SHADE_MODE_TEXTURE
```

Однако, если подключено наложение текстуры, есть возможность выполнять растеризацию только для излучательного или плоского затенения, поэтому затенение по Гуро включать нельзя. Например, чтобы создать в 16-битовом режиме ландшафт с затенением по Гуро и без текстуры, следует воспользоваться следующими константами.

```
(POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
```

Ниже представлены константы для создания ландшафта с плоским затенением и текстурой.

```
(POLY4DV2_ATTR_RGB16 |  
POLY4DV2_ATTR_SHADE_MODE_FLAT |  
POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
```

Как видите, атрибуты многоугольника комбинируются с помощью побитового оператора ИЛИ. Теперь поговорим о поддержке текстурных карт.

Обратите внимание на рис. 10.27. На нем представлен битовый образ, который можно использовать для наложения текстуры на ландшафт (он сгенерирован как текстурная карта ландшафта, поэтому работает вполне удовлетворительно). Размеры этого битового образа — 256x256, но он может быть квадратом, сторона которого равна любой степени двойки, начиная с 8x8 пикселей. Размер 256x256 — максимальный из тех, что поддерживается в нашей системе для единой текстуры на стадии растеризации.

Я использовал подход, состоящий в том, что сначала вычислялись координаты текстуры для входной карты, а затем текстура накладывалась на список вершин карты ландшафта. Конечно же, можно было воспользоваться огромными текстурными картами, разбить их на фрагменты размером 256x256 пикселей, а затем наложить на определенные многоугольники. Однако сейчас все происходит проще, и я не стремлюсь к созданию настоящего процессора ландшафтов. Я просто хочу создать что-нибудь пригодное для демонстрации отсечения. Сейчас, как только выбрана карта текстуры, в функцию текстуры передается имя файла, в котором содержится эта карта (вместе с именем файла, содержащим 256-цветовой битовый образ, из которого будет извлекаться информация о высотах), и в функции будут вычислены координаты текстуры, которой "покрывается"

ландшафт. Это все довольно просто, но на практике возникает множество **деталей**, связанных с шириной, высотой и количеством **ячеек**, содержащихся в карте ландшафта.

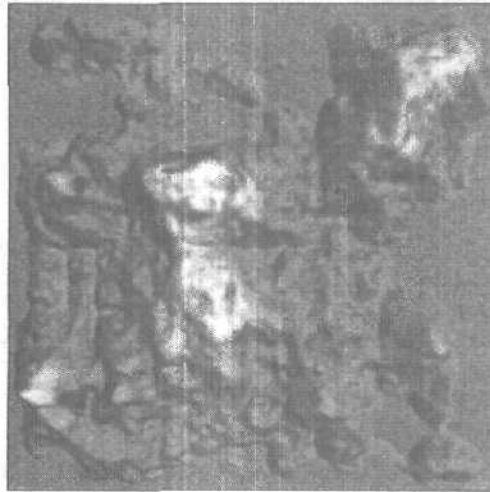


Рис. 10.27. Пример текстуры ландшафта

Таким образом, если на ландшафт накладывается текстура, то в **функцию** необходимо передать имя файла, **содержащего** 16-битовую карту текстуры. Конечно же, эта карта должна быть квадратной. Функция накладывает текстуру на ландшафт, вычисляя координаты текстуры, отображаемые на каждый треугольник, входящий в состав ячеек, из которых состоит ландшафт.

Вот и все, что нужно знать о функции, генерирующей ландшафты. В большинстве случаев на вход этой функции подается два файла. Один из них содержит карту высот в 256-цветовом формате **.BMP**. По возможности **эта** карта должна быть квадратом, и обычно имеет размеры до 40х40 пикселей. Дело в том, что карта больших размеров чрезмерно перегрузит наш небольшой программный игровой процессор.

Теперь приведем саму функцию, а после рассмотрим, как она вызывается в различных ситуациях.

```
int Generate_Terrain_OBJECT4DV2(
    OBJECT4DV2_PTR obj,      // Указатель на объект
    float twidth,             // Ширина в мировых координатах
                              // (по оси x)
    float theight,            // Длина в мировых координатах
                              // (по оси z)
    float vscale,             // Масштаб ландшафта по
                              // вертикали
    char *height_map_file,    // Имя файла с 256-цветной
                              // картой высот
    char *texture_map_file,   // Имя файла с картой текстуры
    int rgbcolor,             // Цвет ландшафта при
                              // отсутствии текстуры
    VECTOR4D_PTR pos,         // Начальное положение
    VECTOR4D_PTR rot,         // Начальная ориентация
    int poly_attr)            // Атрибуты затенения
{
```

```

// Эта функция генерирует ландшафт с заданными шириной и
// длиной в плоскости xz. Ландшафт определяется полем
// высот, которое кодируется с помощью индексов цветов
// 256-цветовой текстуры. Индекс 0 соответствует
// базовому уровню, а индекс 255 -уровню 1.0. Над этим
// диапазоном высот осуществляется масштабное
// преобразование путем умножения высоты каждой точки на
// множитель vscale. Сгенерированное поле высот будет
// состоять из треугольников, вершины которых находятся
// в каждой точке, в которой заданы высоты. Таким
// образом, если карта высот имеет размеры 256 x 256
// точек, то в результате получится каркас, состоящий из
// (256-1) x (256-1) многоугольников, и размер его в
// мировой системе координат равен twidth x theight
// (в плоскости xz). Если задан файл с картой текстуры,
// он накладывается на ландшафт, и генерируются
// координаты текстуры. Функция генерирует ландшафты
// либо в 8-битовом, либо в 16-битовом режиме.
// Убедитесь, что используются правильные атрибуты и
// битовая глубина текстурной карты!!

```

```
char buffer[256]; // Рабочий буфер.
```

```
float col_tstep, row_tstep;
float col_vstep, row_vstep;
int columns, rows;
```

```
int rgbwhite;
```

```
BITMAP_FILEheight_bitmap; //Хранилище карты высот
```

```
// Этап 1: очистка и инициализация объекта
memset(obj, 0, sizeof(OBJECT4DV2));
```

```
// Устанавливается, что объект активный и видимый
obj->state = OBJECT4DV2_STATE_ACTIVE |
OBJECT4DV2_STATE_VISIBLE;
```

```
// Задается положение объекта.
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;
```

```
// Создается надлежащее слово цвета на основе выбранной
// битовой глубины ландшафта. Цвет всегда задается в
// RGB-формате 5.6.5, поэтому его нужно преобразовать
// в 8-битовый формат
if (poly_attr & POLY4DV1_ATTR_8BITCOLOR)
{
    rgbcolor = rgblookup[rgbcolor];
    rgbwhite = rgblookup[RGB16Bit(255,255,255)];
} // if
else
```

```

    |
    rgbwhite = RGB16Bit(255,255,255) ;
} // else

// Задаются количество каркасов
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;

// Очищается битовый образ
memset(&height_bitmap, 0, sizeof(BITMAP_FILE));
memset(&bitmap16bit, 0, sizeof(BITMAP_FILE));

// Этап 2: загрузка поля высот
Load_Bitmap_File(&height_bitmap, height_map_file);

// Вычисляются базовые данные
columns = height_bitmap.bitmapinfoheader.biWidth;
rows = height_bitmap.bitmapinfoheader.biHeight;

col_vstep = twidth / (float)(columns - 1);
row_vstep = theight / (float)(rows - 1);

sprintf(obj->name, "Terrain:%s%s", height_map_file,
        texture_map_file);
obj->num_vertices = columns * rows;
obj->num_polys = ((columns - 1) * (rows - 1) ) * 2;

// Сохраняются некоторые результаты, чтобы облегчить
// дальнейшее использование ландшафта
// Использование вспомогательных переменных в объекте
// также может оказаться хорошим приемом!
obj->ivar1 = columns;
obj->ivar2 = rows;
obj->fvar1 = col_vstep;
obj->fvar2 = row_vstep;

// Выделяется память для вершин и многоугольников. В
// данном случае некоторые параметры лишние, но это
// не важно
if (!Init_OBJECT4DV2(obj, // Объект, для которого
                    // выделяется память.
                    obj->num_vertices,
                    obj->num_polys,
                    obj->num_frames))
    |
    Write_Error("\nTerrain generator error"
                "(can't allocate memory).");
} // if

// Загружается карта текстуры (при ее наличии)
if ( (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE) &&

```

```

texture_map_file)
// Загружается текстура с диска
Load_Bitmap_File(&bitmap16bit, texture_map_file);

// Задается нужный размер и битовая глубина
obj->texture =
    (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));
Create_Bitmap(obj->texture, 0, 0,
    bitmap16bit.bitmapinfoheader.biWidth,
    bitmap16bit.bitmapinfoheader.biHeight,
    bitmap16bit.bitmapinfoheader.biBitCount);

// Задается битовый образ (позже это будет
// реализовано в 8-битовом и 16-битовом режимах)
if (obj->texture->bpp == 16)
    Load_Image_Bitmap16(obj->texture,
        &bitmap16bit, 0, 0,
        BITMAP_EXTRACT_MODE_ABS);
else
{
    Load_Image_Bitmap(obj->texture,
        &bitmap16bit, 0, 0,
        BITMAP_EXTRACT_MODE_ABS);
} // else 8 bit

// Вычисляются коэффициенты дискретизации текстурной
// карты для вычисления координат текстуры
col_tstep =
    (float)(bitmap16bit.bitmapinfoheader.biWidth-1)/
    (float)(columns - 1);
row_tstep =
    (float)(bitmap16bit.bitmapinfoheader.biHeight-1)/
    (float)(rows - 1);

// Для объекта задается флаг, соответствующий
// наличию текстуры
SET_BIT(obj->attr, OBJECT4DV2_ATTR_TEXTURES);

// Готово; поэтому выгружается битовый образ
Unload_Bitmap_File(&bitmap16bit);
} // if

Write_Error("\ncolumns = %d, rows = %d", columns, rows);
Write_Error("\ncol_vstep = %f, row_vstep = %f",
    col_vstep, row_vstep);
Write_Error("\ncol_tstep = %f, row_tstep = %f", col_tstep,
    row_tstep);
Write_Error("\nnum_vertices = %d, num_polys = %d",
    obj->num_vertices, obj->num_polys);

// Этап 4: создание списка вершин и списка координат
// текстуры в виде строк

```

```

for(int curr_row = 0; curr_row < rows; curr_row++)
{
    for(int curr_col = 0; curr_col < columns; curr_col++)
    {
        int vertex = (curr_row * columns) + curr_col;
        // Вычисляются параметры вершины.
        obj->vlist_local[vertex].x =
            curr_col * col_vstep - (twidht/2);
        obj->vlist_local[vertex].y = vscale *
            ((float)height_bitmap.buffer[curr_col +
                (curr_row * columns)]) / 255;
        obj->vlist_local[vertex].z =
            curr_row * row_vstep - (theight/2);

        obj->vlist_local[vertex].w = 1;

        // Для каждой вершины задается, по крайней мере,
        // положение. Это указывается с помощью флагов
        SET_BIT(obj->vlist_local[vertex].attr,
            VERTEX4DTV1_ATTR_POINT);

        // Нужны ли координаты текстуры?
        if ((poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
            && texture_map_file)
        {
            // Теперь координаты текстуры
            obj->tlist[vertex].x = curr_col * col_tstep;
            obj->tlist[vertex].y = curr_row * row_tstep;
        }
    } // if

    Write_Error("\nVertex %d: V[%f, %f, %f], "
        " T[%f, %f]", vertex,
        obj->vlist_local[vertex].x,
        obj->vlist_local[vertex].y,
        obj->vlist_local[vertex].z,
        obj->tlist[vertex].x,
        obj->tlist[vertex].y);
} // for curr_col

} // for curr_row

// Производится ли поворот?

// Вычисляются средний и максимальный радиусы
Compute_OBJECT4DV2_Radius(obj);

Write_Error("\nObject average radius - %f, "
    "max radius = %f",
    obj->avg_radius[0], obj->max_radius[0]);

// Этап 5: создание списка многоугольников
for (int poly=0; poly < obj->num_polys/2; poly++)
{

```

```

// Многоугольники создаются по общему образцу, по 2
// треугольника в каждом четырехугольнике. Следует
// хорошо понимать, как правильно задавать индексы.
// Суть в том, что есть массив вершин  $m \times n$ , и на его
// основе создается список многоугольников
//  $(m-1) \times (n-1)$ , где каждому квадрату соответствуют
// 2 треугольника. Один из способов нумерации
// многоугольников - "змейкой". Другой способ -
// воспользоваться двумя циклами. Возможны и другие
// методы нумерации многоугольников

int base_poly_index = (poly % (columns-1)) +
    (columns * (poly / (columns - 1)));

// Левый верхний многоугольник
obj->plist[poly*2].vert[0]=base_poly_index;
obj->plist[poly*2].vert[1]=base_poly_index+columns;
obj->plist[poly*2].vert[2]=base_poly_index+columns+1;

// Правый нижний многоугольник
obj->plist[poly*2+1].vert[0]=base_poly_index;
obj->plist[poly*2+1].vert[1]=base_poly_index+columns+1;
obj->plist[poly*2+1].vert[2]=base_poly_index+1;

// Указываем в списке вершин многоугольника на
// список вершин объекта. Заметим, что это излишне,
// поскольку список многоугольников в данном случае
// содержится в объекте и пользователь может
// выбирать, какой список вершин применяется для
// построения геометрии из многоугольников -
// локальный или преобразованный. Возможно, для
// многоугольников, являющихся частью объекта, здесь
// лучше было бы задать значение NULL
obj->plist[poly*2].vlist = obj->vlist_local;
obj->plist[poly*2+1].vlist = obj->vlist_local;

// Задаются атрибуты многоугольника, переданные в
// функцию
obj->plist[poly*2].attr = poly_attr;
obj->plist[poly*2+1].attr = poly_attr;

// Выполняется некоторая проверка, чтобы убедиться,
// что все вспомогательные данные заданы как следует

// Задается цвет многоугольника
obj->plist[poly*2].color = rgbcolor;
obj->plist[poly*2+1].color = rgbcolor;

// Проверяется, обладает ли многоугольник затенением
// по Гуро. Если это так, в его вершинах нужно
// вычислять нормали
if ( (obj->plist[poly*2].attr &
    POLY4DV2_ATTR_SHADE_MODE_GOURAUD) ||
    (obj->plist[poly*2+1].attr &

```

```

    POLY4DV2_ATTR_SHADE_MODE_PHONG) )
{
    // Нормали нужно задавать во всех вершинах
    // данного треугольника; указываем это с помощью
    // флагов
    SET_BIT(obj->
        vlist_local[obj->plist[poly*2].vert[0]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->
        vlist_local[obj->plist[poly*2].vert[1]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->
        vlist_local[obj->plist[poly*2].vert[2]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);

    SET_BIT(obj->
        vlist_local[obj->plist[poly*2+1].vert[0]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->
        vlist_local[obj->plist[poly*2+1].vert[1]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->
        vlist_local[obj->plist[poly*2+1].vert[2]]
        .attr, VERTEX4DTV1_ATTR_NORMAL);

} // if

// Если текстура задана, подключаются координаты
// текстуры
if (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
    // К текущему многоугольнику применяется
    // текстура
    obj->plist[poly*2].texture = obj->texture;
    obj->plist[poly*2+1].texture = obj->texture;

    // Присваиваются координаты текстуры для
    // левого верхнего многоугольника
    obj->plist[poly*2].text[0]=base_poly_index;
    obj->plist[poly*2].text[1]=base_poly_index
        +columns;
    obj->plist[poly*2].text[2]=base_poly_index
        +columns+1;

    // Правый нижний многоугольник
    obj->plist[poly*2+1].text[0]=base_poly_index;
    obj->plist[poly*2+1].text[1]=base_poly_index+
        columns+1;
    obj->plist[poly*2+1].text[2]=base_poly_index+1;

    // Perezapisывaется базовый цвет, чтобы учесть
    // отражение
    obj->plist[poly*2].color = rgbwhite;
    obj->plist[poly*2+1].color = rgbwhite;
}

```

```

// Задаются атрибуты координат текстуры
SET_BIT(obj->
    vlist_local[obj->plist[poly*2].vert[0]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);
SET_BIT(obj->
    vlist_local[obj->plist[poly*2].vert[1]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);
SET_BIT(obj->
    vlist_local[obj->plist[poly*2].vert[2]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);

SET_BIT(obj->
    vlist_local[obj->plist[poly*2+1].vert[0]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);
SET_BIT(obj->
    vlist_local[obj->plist[poly*2+1].vert[1]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);
SET_BIT(obj->
    vlist_local[obj->plist[poly*2+1].vert[2]]
    .attr, VERTEX4DTV1_ATTR_TEXTURE);

} // if

// Задается режим материала для эмуляции версии 1.0
SET_BIT(obj->plist[poly*2].attr,
    POLY4DV2_ATTR_DISABLE_MATERIAL);
SET_BIT(obj->plist[poly*2+1].attr,
    POLY4DV2_ATTR_DISABLE_MATERIAL);

// Наконец, устанавливается, что многоугольник
// активен
obj->plist[poly*2].state = POLY4DV2_STATE_ACTIVE;
obj->plist[poly*2+1].state = POLY4DV2_STATE_ACTIVE;

// Указываем в списке вершин многоугольника на
// список вершин объекта. Заметим, что это излишне,
// поскольку список многоугольников в данном случае
// содержится в объекте и пользователь может
// выбирать, какой список вершин применяется для
// построения геометрии из многоугольников -
// локальный или преобразованный. Возможно, для
// многоугольников, являющихся частью объекта, здесь
// лучше было бы задать значение NULL
obj->plist[poly*2].vlist = obj->vlist_local;
obj->plist[poly*2+1].vlist = obj->vlist_local;

// Задается список координат текстуры; это
// необходимое действие
obj->plist[poly*2].tlist = obj->tlist;
obj->plist[poly*2+1].tlist = obj->tlist;

} // for poly
#endif

```

```

for (poly=0; poly < obj->num_polys; poly++)
{
    Write_Error("\nPoly %d: Vi[%d, %d, %d], "
        "Ti[%d, %d, %d]",
        poly, obj->plist[poly].vert[0],
        obj->plist[poly].vert[1],
        obj->plist[poly].vert[2],
        obj->plist[poly].text[0],
        obj->plist[poly].text[1],
        obj->plist[poly].text[2]);
} // end
tfendif

// Вычисляются длины нормалей к многоугольникам
Compute_OBJECT4DV2_Poly_Normals(obj);

// Вычисляются нормали в вершинах для многоугольников с
// затенением по Гуро
Compute_OBJECT4DV2_Vertex_Normals(obj);

// Код успешного выполнения
return(1);
} // Generate_Terrain_OBJECT4DV2

```

В функции используется загрузчик файлов в формате **.PLG**. Было решено начать именно с этого формата, поскольку в нем имеется все необходимое для создания объекта **OBJECT4DV2**. Теперь рассмотрим, как вызывается эта функция. Ниже приведен пример ее вызова, с помощью которого генерируется ландшафт 32x32 с наложением текстуры, плоским затенением и размерами в мировой системе координат 4000x4000. Для этого ландшафта используется карта высот, содержащаяся в файле **EARTHHEIGHTMAP01.BMP**, и карта текстуры (256x256x16), содержащаяся в файле **EARTHCOLORMAP01.BMP**.

VECTOR4D terrain\_pos - {0,0,0,0};

```

Generate_Terrain_OBJECT4DV2(
    &obj_terrain, // Указатель на объект
    4000,         // Ширина (по оси x) в
                // мировых координатах
    4000,         // Длина (по оси z) в
                // мировых координатах
    700,          // Масштабный множитель
                // по верти кали
    "earthheightmap01.bmp", // Имя файла с картой
                // высот, закодированных
                // с помощью 256 цветов
    "earthcolormap01.bmp", // Имя файла с картой
                // текстуры
    RGB16Bit(255,255,255), // Цвет ландшафта при
                // отсутствии текстуры
    &terrain_pos, // Начальное положение
    NULL,         // Начальная ориентация
    POLY4DV2_ATTR_RGB16 |
    POLY4DV2_ATTR_SHADE_MODE_FLAT |
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE);

```

Параметры этой функции вполне понятны. В нее нужно передать ширину и даину карты ландшафта, заданные в мировой системе координат (это будут границы в плоскости  $xz$ ; в данном случае заданы значения  $4000 \times 4000$ ), и масштабный множитель карты ландшафта по высоте. На него будут умножаться все высоты, поэтому он представляет максимальную *высоту*, которая может *быть* достигнута на данном ландшафте (в нашем случае задано значение 700). Далее следуют имена файлов на диске, *содержащих* карты высот и текстуры (если текстура отсутствует, вместо файла текстурной карты должно быть задано значение NULL). Затем идет цвет ландшафта (если на ландшафт не накладывается текстура, функции нужно указать его цвет). После этого задается положение ландшафта в мировой системе координат (обычно подходят координаты 0,0,0). Далее следует начальная ориентация каркаса (если обработка этого параметра *еще* не реализована, задайте значение NULL), и *наконец* — атрибуты многоугольников (в нашем случае это плоское затенение, 16-битовый режим и наличие текстуры).

Для ландшафта можно создать любое количество объектов. При желании можно создать один огромный участок размером, скажем,  $16 \times 16$  объектов, а затем воспользоваться преимуществами отбраковки объектов. Но, как уже упоминалось, в мои намерения входило создание какого-нибудь объекта, состоящего из большого количества *многоуголь-*ников и *моделирующего* фрагмент окружающей или внутренней обстановки, по *которо-*му можно было бы перемешаться, и эта цель достигнута.

## Генерация данных ландшафта

Возможно, представленный выше метод и является замечательным. Но как *самому* сгенерировать данные для ландшафта? Для этого можно создать специальную программу. Кроме того, следует позаботиться о том, чтобы каждая текстура, которая накладывается на ландшафт, выглядела реалистично. Например, снеговые шапки должны находиться на вершинах гор, а озера должны быть синего цвета. Большинство создателей игр для генерации ландшафтов пользуются программой, которая может также сгенерировать карту текстуры для ландшафта. Одна из таких программ называется *VistaPro*; если я не ошибаюсь, уже существует ее версия 4.0. Проблема в том, что не совсем понятно, кому принадлежит лицензия на *нее*, поэтому лучше всего зайти в *Internet* и самостоятельно осуществить поиск по ключевому слову *VistaPro*. Еще одна программа, *генерирующая* ландшафты и карты высот, — *Bryce* компании MetaCreations. Однако эта компания была продана, и теперь право собственности на программу принадлежит компании Corel. Впрочем, в *Internet* имеется масса бесплатных программ, предназначенных для генерации ландшафтов.

## Демонстрационная программа

Теперь скомпилируем вместе сгенерированный массив и отсечение и посмотрим, как все это работает. Именно так была создана небольшая демонстрационная программа *DEMO110\_2.CPP|EXE* (8-битовая версия этой программы находится в файле *DEMO110\_2\_8b.CPP|EXE*). Она сделана в виде *симулятора* езды на вездеходе по пустынному острову. Снимки экрана, полученные в процессе работы этой программы, приведены на рис. 10.28 и рис. 10.29. Чтобы скомпилировать эту программу, понадобится основной файл в формате .CPP и библиотека *T3DLIB1-8.CPP|H*, а также стандартные библиотечные файлы DirectX.

Демонстрационная программа, как обычно, работает очень просто. В ней один раз вызывается функция, генерирующая ландшафт с уже рассмотренными параметрами. Затем камера помещается так, чтобы обзор производился из кабины водителя, и вездеход готов к путешествию! По поводу того, как возникает ландшафт, мне добавить нечего.

Информация об управляющих клавишах для этой программы отображается на экране в справке, которую можно включать или отключать с помощью клавиши <H>. Самая

важная **ВОЗМОЖНОСТЬ** — включение и отключение трехмерного отсечения разными плоскостями, которое **осуществляется** с помощью клавиш <X>, <Y> и <Z>. Затем можно поехать вокруг, пытаясь одновременно следить за статистической информацией, отображаемой в нижней части экрана. Обратите внимание, как меняются эти данные в зависимости от того, производится отсечение по различным осям или нет.

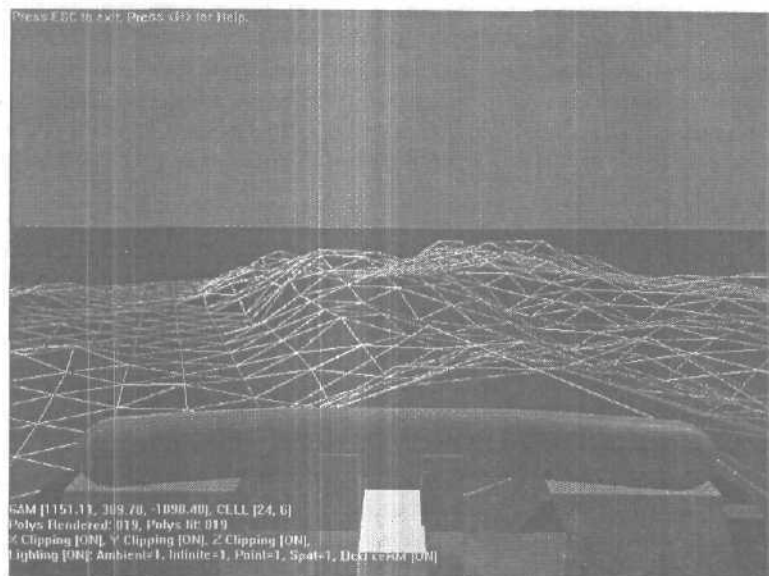


Рис. 10.28. Копия экрана демонстрационной программы в **каркасном режиме**

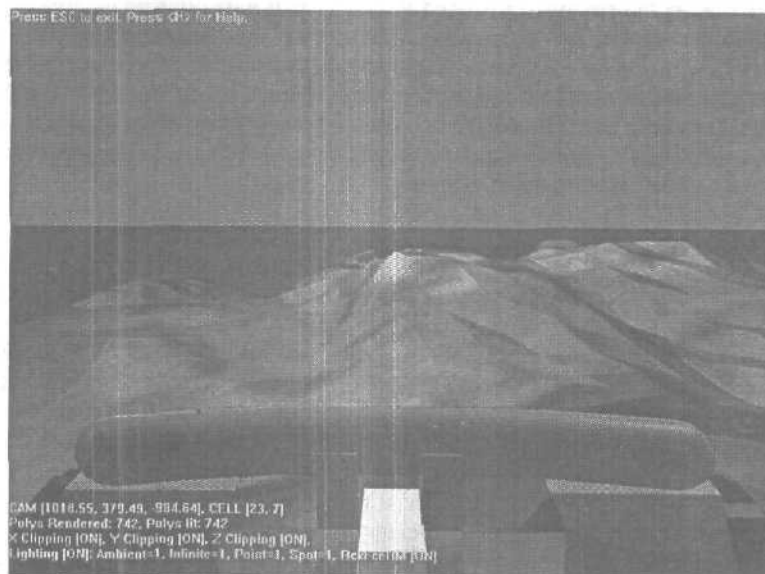


Рис. 10.29. Копия экрана демонстрационной программы в режиме **заполнения с текстурой**

Отключение отсечения плоскостями, перпендикулярными оси z, может привести к аварийной ситуации, поскольку обрабатываются и проецируются довольно большие многоугольники. Конечно же, это неприятно, но компьютеру никакого физического вреда причинено не будет! Чтобы увидеть все многоугольники, попробуйте переключиться в каркасный режим с помощью клавиши <W>.

## Алгоритм езды по ландшафту

Алгоритм, с помощью которого имитируется езда по ландшафту, и простая физическая модель вездехода связаны между собой, поэтому их удобно обсудить вместе. Задача формулируется следующим образом: имеется камера, установленная там, где должен находиться водитель (в ее поле зрения попадает небольшая передняя часть кабины вездехода). Нужно сделать так, чтобы игрок смог водить вездеход по местности, "не погружаясь" **внутрь**, поэтому следует определить, на какой высоте находится тот участок, на котором находится камера, и соответствующим образом изменять ее положение (рис. 10.30).

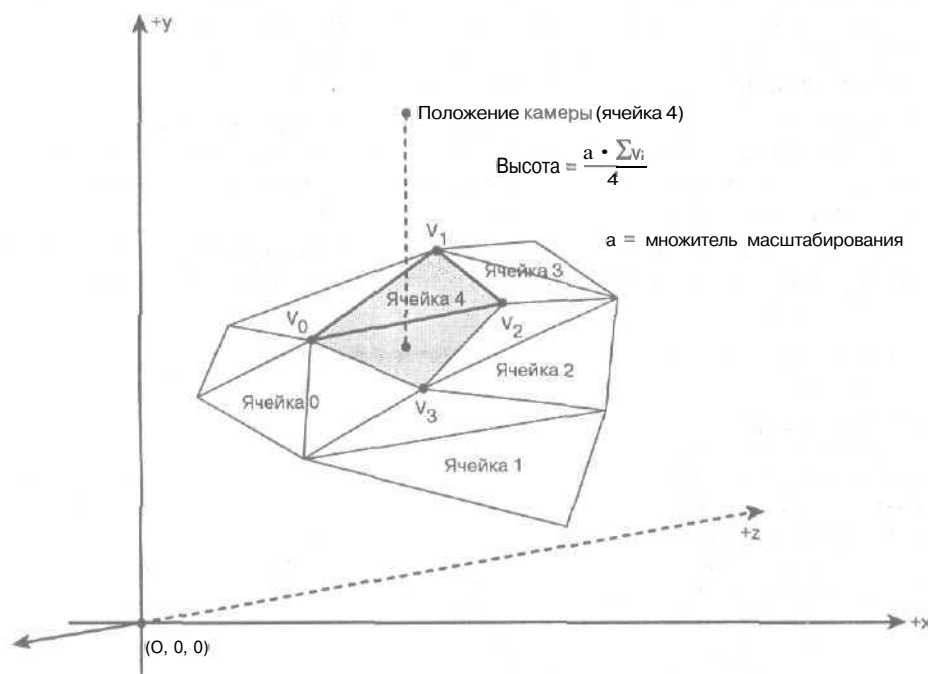


Рис. 10.30. Схема отслеживания неровностей ландшафта

Существует несколько подходов к этой проблеме. Можно было бы разработать реалистическую физическую модель движения вездехода по ландшафту, однако это было бы слишком для такого простого случая. Можно было бы впасть в другую крайность и просто перемещать камеру, помещая ее на высоту, определяемую той ячейкой, где она находится. Я решил поискать золотую середину и воспользовался очень простой физической моделью, основанной на скорости и ускорении, с которыми движется камера, а также силе притяжения. Далее камера и вездеход — это одно и то же. Перечислим основные особенности модели.

- Движение вездехода характеризуется скоростью и направлением, а управлять им можно с помощью **клавиш** со стрелками (при нажатии клавиш со стрелками вверх или вниз вездеход сначала движется с постоянным ускорением вперед или назад, а затем, по достижении определенной скорости — с этой скоростью).
- На вездеход действует постоянная сила притяжения. В результате, если движение вездехода сопровождается снижением высоты (координаты  $y$ ), его скорость возрастает.
- На вездеход действует сила, направленная вверх. Если машина опускается ниже уровня **моря**, то благодаря этой силе она выталкивается на поверхность.
- В каждый момент времени с помощью текущего положения вездехода в мировых координатах определяется, над какой ячейкой ландшафта он расположен, а затем высоты четырех вершин усредняются. Найденное значение высоты сравнивается с текущим, и если разность отрицательная (с учетом дорожного просвета), вертикальная **составляющая** скорости вездехода возрастает. Ускорение, приобретаемое вездеходом, пропорционально разности высот. Кроме того, **существует** верхний предел вертикальной составляющей скорости, поэтому, если вездеходу встречается уступ или значительный перепад высот, его вертикальная координата изменяется скачком и не происходит постепенное отслеживание геометрии.
- Изменение высоты нахождения вездехода над уровнем моря в результате его перемещения по местности сопровождается изменением расстояния от камеры до поверхности ландшафта. Это создает иллюзию того, что при езде по пересеченной местности вездеход испытывает качку.
- Наконец, в код помещен фрагмент, стабилизирующий физическую модель. Поэтому, если положение и ориентация изменяется незначительно, то колебания вездехода затухают.

Приведем некоторые физические константы и переменные, которые используются в ходе симуляции.

```
// Параметры ландшафта
#define TERRAIN_WIDTH 4000
#define TERRAIN_HEIGHT 4000
#define TERRAIN_SCALE 700
#define MAX_SPEED 20

// Параметры физической модели. Изменяя эти константы, можно
// управлять поведением вездехода
float gravity = -.40; // Общая сила гравитации
float vel_y = 0; // Составляющая скорости камеры
// (джипа) по оси y
float cam_speed = 0; // Скорость камеры (джипа)
float sea_level = 50; // Уровень моря
float gclearance = 75; // Зазор между камерой и
// поверхностью ландшафта.
float neutral_pitch = 10; // Зазор в нейтральном состоянии
```

Ниже представлен **код**, в котором генерируется ландшафт и имитируется езда на вездеходе по этому ландшафту.

```
// Имитация движения

// Отслеживаем неровности ландшафта. Просто определяем, над
```

```

// какой ячейкой находится вездеход, затем по индексу в
// списке вершин находим 4 вершины, образующие эту ячейку, и
// усредняем их высоты. На основе соотношения между текущим
// значением высоты и высотой ландшафта вездеход приобретает
// ускорение, направленное вверх или вниз

// В процессе генерации ландшафта некоторые результаты
// сохраняются, чтобы можно было отслеживать неровности
// поверхности
//ivar1 = columns;
//ivar2 = rows;
//fvar1 = col_vstep;
//fvar2 = row_vstep;

int cell_x = (cam.pos.x + TERRAIN_WIDTH/2) /
              obj_terrain.fvar1;
int cell_y = (cam.pos.z + TERRAIN_HEIGHT/2) /
              obj_terrain.fvar1;

static float terrain_height, delta;

// Проверяем, находится ли вездеход на ландшафте
if ( (cell_x >= 0) && (cell_x < obj_terrain.ivar1) &&
     (cell_y >= 0) && (cell_y < obj_terrain.ivar2) )
{
    // Определяем индексы вершин текущего четырехугольника
    int v0 = cell_x + cell_y * obj_terrain.ivar2;
    int v1 = v0 + 1;
    int v2 = v1 + obj_terrain.ivar2;
    int v3 = v0 + obj_terrain.ivar2;

    // Теперь просто обращаемся по этому индексу к таблице
    terrain_height = 0.25 * (obj_terrain.vlist_trans[v0].y +
                             obj_terrain.vlist_trans[v1].y +
                             obj_terrain.vlist_trans[v2].y +
                             obj_terrain.vlist_trans[v3].y);

    // Вычисляем разность высот
    delta = terrain_height - (cam.pos.y - gclearance);

    // Проверяем, не произошло ли проникновение
    if (delta > 0)
    {
        // Тут же применяем силу к камере
        // (имитируем действие рессор)
        vel_y += (delta * (0.025));

        // Проверяем, не произошло ли проникновение. Если
        // вездеход находится ниже уровня поверхности, сразу
        // сдвигаем его вверх, чтобы не зарыться в почву
        cam.pos.y += (delta * .3);

        // Это больше похоже на трюк, чем на физическую модель;
    }
}

```

```

// на основе скорости вездехода и градиента высот
// организуем небольшую качку камеры
cam.dir.x -= (delta*.015);

} // if

} // if

// Замедление камеры
if (cam_speed > (0.25) ) cam_speed-=.25;
else
if (cam_speed < (-0.25) ) cam_speed+=.25;
else
cam_speed = 0;

// Озвучивание игры
DSound_Set_Freq(car_sound_id,8000+fabs(cam_speed)*250);

// Вынуждаем камеру находить стабильную ориентацию
if (cam.dir.x > (neutral_pitch+0.3)) cam.dir.x -= (0.3);
else
if (cam.dir.x < (neutral_pitch-0.3)) cam.dir.x += (0.3);
else
cam.dir.x=neutral_pitch;

// Применение гравитации
vel_y+=gravity;

// Проверяем, не опустился ли вездеход ниже уровня моря,
// и при необходимости выталкиваем его на поверхность
if (cam.pos.y < sea_level)
{
vel_y = 0;
cam.pos.y = sea_level;
} // if

// Сдвигаем камеру
cam.pos.x += cam_speed*Fast_Sin(cam.dir.y);
cam.pos.z += cam_speed*Fast_Cos(cam.dir.y);
cam.pos.y += vel_y;

```

Приведенный выше код очень короткий. В систему вводятся только параметры, описывающие текущее положение камеры, а далее она продолжает функционировать самостоятельно.

Вот и все, что следует знать по поводу этой небольшой демонстрационной программы. Изменение значений различных переменных влияет на поведение транспортного средства. Варьируя параметры физической модели, можно получить все, что угодно, — от танка до морского судна весом в 1 000 000 тонн. Кроме того, затратив совсем немного усилий, вы получите возможность создавать ландшафты с помощью объектов, представляющих собой различные участки местности, и реализовывать трехмерные ландшафтные игры.

## Резюме

Надеюсь, что вы хорошо усвоили алгоритмы отсечения и поняли, что они совсем не сложные, просто нужно внимательно отнестись к некоторым деталям. Кроме того, в игровой процессор добавлена отбраковка, которая исключает из рассмотрения многоугольники, полностью выходящие за рамки области обзора. Именно этот вид отбраковки (а не удаление обратных поверхностей) существенно повышает производительность программы. В результате процессор стал работать еще быстрее. Кроме того, мы получили возможность немного поэкспериментировать с ландшафтами и создали небольшую демонстрационную программу, которую легко переделать в игру, имитирующую прогулку на водном транспортном средстве или езде по пересеченной местности. Конечно, эту игру еще нельзя будет сравнивать с такими играми, как *Splashdown* или *V-Rally*, однако это только начало! Наконец, сделаем еще одно важное замечание: это последняя глава, в которой поддерживается 8-битовый режим.



# ГЛАВА 11

## Организация буфера глубины и видимость

### В этой главе...

• Введение в буфер глубины и определение видимости	958
• Основные принципы работы с Z-буфером	961
• Создание системы Z-буфера	975
• Возможные способы оптимизации Z-буфера	998
• Проблемы, связанные с Z-буфером	1001
• Программное обеспечение и демонстрационные программы, использующие Z-буфер	1002

В этой главе рассматриваются основные методы работы с Z-буфером, которые служат для определения того, является ли видимым данный фрагмент поверхности, и удаления невидимых фрагментов. Как обычно, наш игровой процессор придется переписать, добавив в него поддержку этой технологии; также будет разработано несколько демонстрационных программ. Поверьте, что этого будет вполне достаточно. Приведем список основных тем, касающихся данной области:

- буфер глубины и определение видимости;
- основные принципы работы с Z-буфером;
- уравнение плоскости;
- Z-интерполяция;
- 1/z -буферизация;
- создание системы Z-буфера;

- добавление поддержки **Z-буфера** в систему растеризации;
- возможные способы **оптимизации Z-буфера**;
- новое **программное** обеспечение и программы, демонстрирующие работу Z-буфера.

## Буфер глубины и определение видимости

К сожалению, часто происходит путаница двух следующих концепций: удаления скрытых поверхностей и определения того, является ли поверхность видимой.

Дело в том, что в одних случаях это одно и то же, а в других — это разные вещи. Например, когда в игровом конвейере производится удаление обратных поверхностей, при этом определенно выполняется удаление скрытых поверхностей. С другой стороны, отсечение — это и удаление скрытых поверхностей (отсекаются те области, которые не попадают в поле зрения), и вместе с тем определение видимых поверхностей, поскольку нужно определить, что является видимым и не подлежит отсечению. Я сам не могу дать прямого ответа на вопрос о том, что есть что, потому что такого определения не существует. Далее процесс определения того, что нужно выводить на экран, а что — нет, в котором не производится какого-нибудь специфического удаления фрагментов объекта, будет называться *определением видимых поверхностей* (visible surface determination — VSD). Этот термин принят во многих книгах, поэтому он не должен вызывать удивления. Далее, когда речь идет о **Z-буферизации**, независимо от того, какие действия описываются, следует иметь в виду, что алгоритм работает на уровне пикселей, поэтому *здесь* нет поверхностей, а есть только их компоненты или фрагменты! Вывод: принятая в этой области терминология неадекватно отражает суть дела. Такие принципиальные моменты я буду комментировать и далее.

Если вы дочитали книгу до этого места, то вполне в состоянии написать свой собственный трехмерный игровой процессор. Мы уже изложили все основные аспекты визуализации и создания трехмерного конвейера. Конечно же, физические модели и игровые сюжеты выходят за рамки данной книги, поскольку внимание в ней уделялось тому, как преобразовывать, освещать и визуализировать многоугольники. Итак, что мы имеем сейчас? У нас есть достаточно сложная осветительная система, возможность накладывать текстуру и производить отсечение в двумерном и трехмерном пространстве, но нашим моделям из многоугольников и всей окружающей обстановке все еще присуща некоторая упрощенность. На то есть две причины: во-первых, мы еще ни разу не занимались загрузкой уровней с интерьером, поскольку просто не располагаем соответствующим программным инструментом или форматом файлов; тем не менее, технология их визуализации у нас уже разработана. Вторая причина более тонкая. Дело в том, что мы не можем сортировать многоугольники или принимать решение о том, в каком порядке они будут выводиться.

До настоящего времени для сортировки многоугольников использовался алгоритм художника. Это не что иное, как сортировка списка многоугольников по минимальному, среднему или максимальному значению координаты *z*, характеризующему каждый многоугольник. Далее многоугольники просто **выводятся** на экран, начиная от самых удаленных и заканчивая самыми приближенными к игроку. Этот способ напоминает действия художника, когда он пишет полотно. До сих пор такой метод в основном **работал** правильно, однако мы уже знаем, что он обладает рядом недостатков. Например, в нем возникают проблемы при обработке протяженных многоугольников, а иногда возникают просто неоднозначные ситуации (рис. 11.1).

К преимуществам алгоритма художника следует отнести то, что он работает в 99% случаев, с которыми нам приходилось сталкиваться до этого. Он замечательно подходит в ситуациях, когда игровая сцена в основном заполнена объектами (если эти объекты не перекрываются, и каждый из них состоит из небольших многоугольников). Кроме того, метод вполне пригоден для таких каркасов, как модели ландшафтов, которые характеризуются регулярностью и неперекрывающейся геометрией. Другими словами, хотя ландшафт и трехмерный, по большей части его можно считать двумерным за исключением фрагментов, в которых происходит определенный перепад высот.

Однако алгоритм художника (далее мы будем называть его просто *Z-сортировкой* (Z-sorting)) может привести к плохим результатам, когда в игру вступают такие специальные эффекты, как взаимопроникновение, сложная геометрия, а также геометрия, в которой сочетаются большие и маленькие элементы и т.п. Чтобы понять, в каком порядке следует выводить многоугольники в таких ситуациях, понадобится другой метод. Первое, что приходит в голову, — переписать алгоритм Z-сортировки и сделать его “умнее”. На самом деле это реальная задача. Например, можно рассмотреть проблему для двух многоугольников и провести для них тесты, чтобы определить, в правильном ли порядке они выводятся. Алгоритмы этого класса обычно называют *алгоритмами списков с приоритетами* (list priority algorithms). Часто они носят чье-то имя, но обычно это алгоритмы, основанные на обычном здравом смысле. Суть такова: если имеется список многоугольников, отсортированных по координате  $z$ , и нужно убедиться, что в этом списке отсутствуют неправильно отсортированные пары многоугольников, то применяются определенные тесты, помогающие в решении этой задачи. Приведем один из наборов тестов из алгоритма Невеля-Санча (Newell-Sancha).

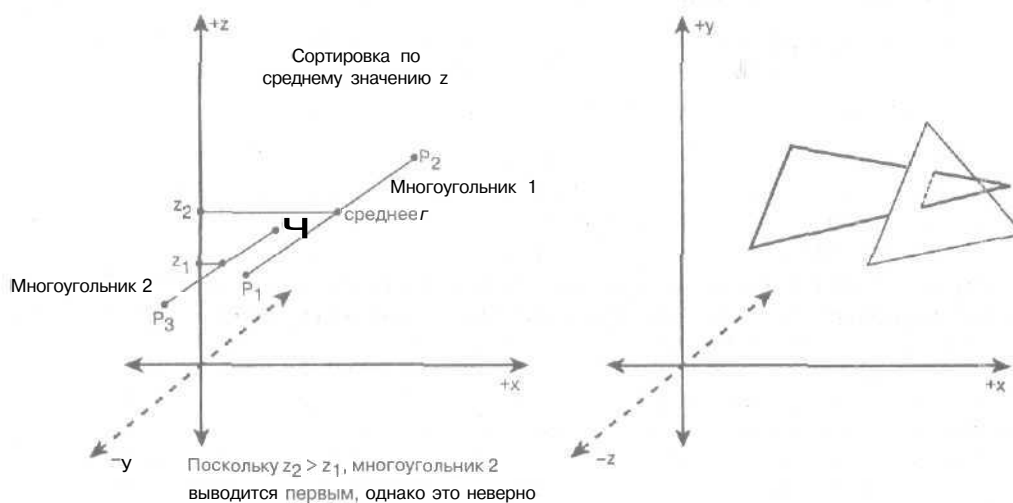


Рис. 11.1. Случай, в котором  $z$ -сортировка не подходит

Рассмотрим многоугольники  $P_1$  и  $P_2$ , изображенные на рис. 11.2. Они отсортированы по глубине в соответствии с максимальным значением координаты  $z$ . Таким образом, первым в списке стоит многоугольник, который расположен дальше всего от точки наблюдения. Далее выполняются такие тесты.

- Тест 1 (проверка на перекрывание по  $z$ ). Если многоугольники  $P_1$  и  $P_2$  неперекрываются по оси  $z$ , то никакая часть многоугольника  $P_1$  не может перекрываться с многоугольником  $P_2$  (другими словами, если  $P_{1_{\min}} > P_{2_{\max}}$ , то многоугольник  $P_1$  заносится в буфер кадров). Этот случай проиллюстрирован на рис. 11.2л. Если же  $P_{1_{\min}} < P_{2_{\max}}$ , то возможны ситуации, когда многоугольник  $P_1$  заслоняет многоугольник  $P_2$  несмотря на то, что при сортировке  $P_1$  получился более удаленным, и должен быть выведен на экран первым. В этом случае необходимо выполнить ряд дополнительных тестов. Как только любой из этих тестов четко определит, заслоняется ли многоугольник  $P_2$  многоугольником  $P_1$ , можно завершать работу алгоритма.
- Тест 2 (проверка с помощью ограничивающего прямоугольника). Проверяем, перекрываются ли расположенные в плоскости  $xy$  прямоугольники, ограничивающие многоугольники  $P_1$  и  $P_2$ . Если перекрытия по осям  $y$  и  $z$  нет, то эти многоугольники не могут закрывать друг друга (рис. 11.2б).
- Тест 3 (проверка разбиением на полупространства). Плоскость, в которой лежит многоугольник  $P_2$ , разбивает все пространство на две части. Проверяем, находится ли многоугольник  $P_1$  в дальнем полупространстве относительно точки наблюдения? Если это так, то выводим многоугольник  $P_1$  и завершаем работу алгоритма (рис. 11.2в).
- Тест 4 (проверка разбиением на полупространства). Проверяем, находится ли многоугольник  $P_2$  в ближнем относительно точки наблюдения полупространстве многоугольника  $P_1$ ? Если это так, выводим многоугольник  $P_2$  и завершаем работу алгоритма.
- Тест 5 (проверка перспективы). Не перекрываются ли аксонометрические проекции многоугольников  $P_1$  и  $P_2$ ? Если не перекрываются, выводим многоугольник  $P_1$  и завершаем работу алгоритма.

Если все тесты дали отрицательный результат, многоугольники  $P_1$  и  $P_2$  меняются местами и помечаются как переставленные. В случае, когда будет предпринята попытка снова их переставить, возможно образование циклического перекрытия (рис. 11.3). Чтобы решить эту проблему, один из многоугольников необходимо разбить на два.

Если вы не поняли, как работает этот алгоритм, не беспокойтесь — им все равно никто не пользуется! Я просто хотел дать вам общее представление о нем. Суть в том, что предпринимается попытка решить проблему, формулируя все более сложные вопросы. Так происходит до тех пор, пока все многоугольники не будут выстроены в правильном порядке. Стоит ли это выстраивание такого объема работы? Конечно же, нет! Если вы хотите получить правильный порядок, то не воспользуйтесь таким запутанным алгоритмом, а если вас не интересует строгий порядок, то вы тем более не станете этого делать, поскольку  $Z$ -сортировка и так в большинстве случаев работает хорошо (во многих играх для персональных компьютеров используется  $Z$ -сортировка и, скорее всего, это совсем незаметно!).

Таким образом, чтобы решить эту проблему и добиться правильного порядка визуализации, нам понадобится что-нибудь позлеегантнее (или попроще), чем все эти запутанные тесты. Итак, знакомьтесь:  $Z$ -буфер!

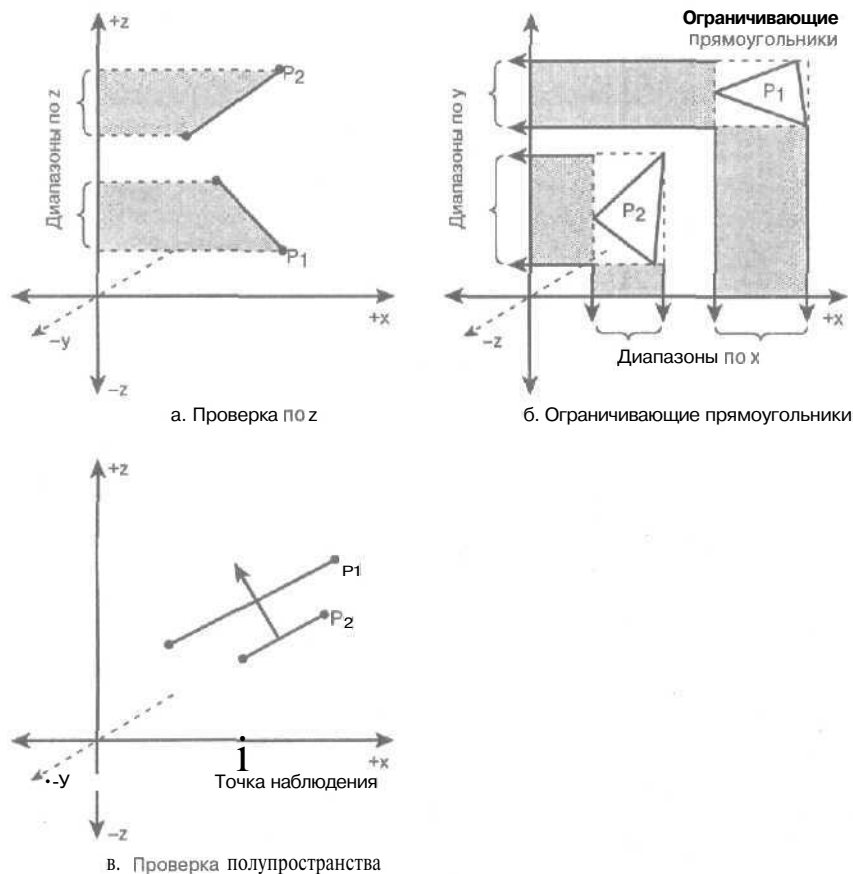


Рис. 11.2. Сортировка многоугольников

## Основные принципы работы с Z-буфером

Как ни странно, решением всех проблем является **Z-сортировка**. Напомним, что она хорошо работает для маленьких многоугольников, но начинает давать сбои, когда их протяженность становится значительной. Причина сбоев заключается в неподходящей сортировке (в чем мы имели возможность убедиться, анализируя работу алгоритма в рассмотренных выше случаях). А что, если Z-сортировка будет производиться для каждого пикселя? В этом случае получить неправильный порядок **расположения** двух многоугольников невозможно, потому что мы опускаемся при этом на самый нижний уровень, возможный для элементов экрана. Именно по такому **принципу** работает **Z-буфер**. В нем производится Z-сортировка на уровне пикселей, т.е. это алгоритм художника на уровне пикселей. Попробуйте повторить это три раза!

Идея настолько проста, что чем-то напоминает обман. Впервые этот алгоритм разработан в 1974 г. Эдвином **Кэтмуллом** (Edwin Catmull) (по крайней мере, нас хотят в этом убедить). Описание алгоритма выглядит очень просто.

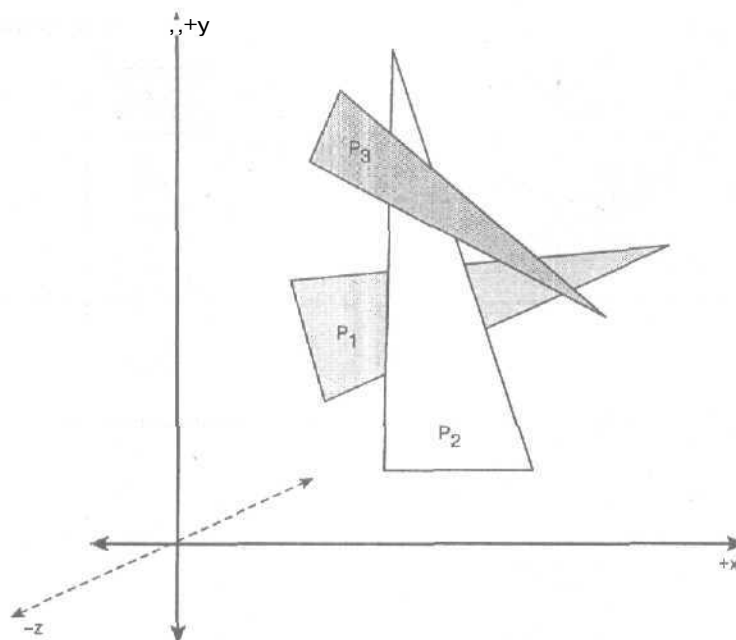


Рис. 11.3. Циклическое перекрытие многоугольников

Для экрана размерами  $M \times N$  создается Z-буфер  $zbuffer[x,y]$ , в котором содержатся значения координаты  $z$  для каждого пикселя каждого многоугольника. Эти значения являются результатом растеризации или сканирования. Z-буфер также имеет размеры  $M \times N$ ; он инициализируется максимально возможными значениями координаты  $z$  (во всех случаях подходит бесконечно большое значение). С помощью Z-буфера производятся следующие действия.

Для каждого многоугольника в списке визуализации  
begin

1. Производим растеризацию многоугольника и генерируем  $x_i, y_i, z_i$  для всех значений, которые возможны при проецировании многоугольника на экран в процессе растеризации.
2. if ( $z_i < zbuffer[x_i, y_i]$ ) then  
    вносим значение  $z_i$  в Z-буфер,  
     $zbuffer[x_i, y_i] = z_i$ , и выводим пиксель на экран,  $Plot(x_i, y_i)$ .

end

Схема проста. Создается массив, состоящий из чисел с плавающей точкой или **целых** чисел, с помощью которого отслеживаются значения координаты  $z$  каждого сканируемого многоугольника. Однако вместо того, чтобы выводить каждый пиксель на экран, значение координаты  $z$  текущего пикселя сравнивается с тем значением, которое находится на этом месте в Z-буфере. Если координата  $z$  текущего сканируемого многоугольника, соответствующая пикселю  $(x_i, y_i)$ , меньше **текущего** значения  $z$ , хранящегося в элементе массива Z-буфера  $zbuffer[x_i, y_i]$ , то это значение обновляется новым значением

$z_1$ , после чего пиксель выводится на экран. В противном случае ничего не происходит. Графическая схема алгоритма приведена на рис. 11.4.

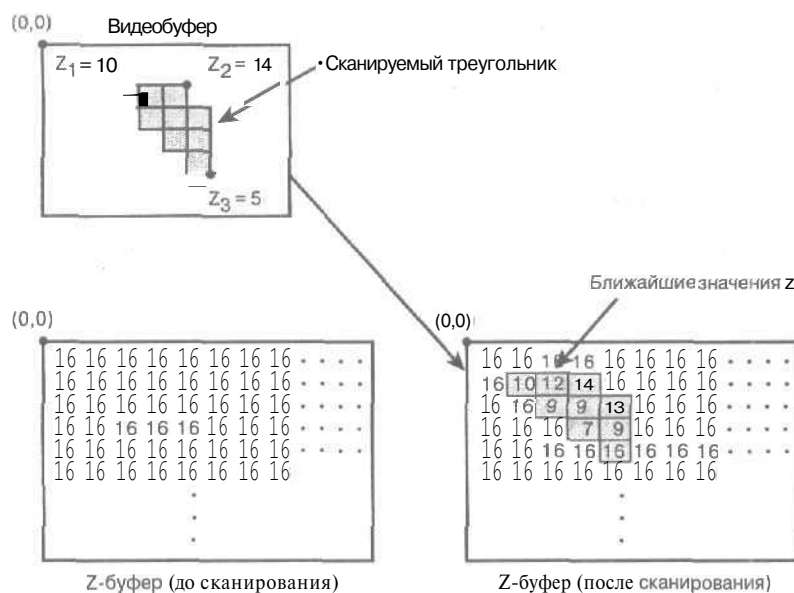


Рис. 11.4. Алгоритм работы Z-буфера

В результате применения этого алгоритма пиксели, соответствующие многоугольникам, которые расположены ближе, перекрывают пиксели более удаленных многоугольников. При этом в двумерном Z-буфере происходит обновление координат  $z$  для всех возможных положений  $(x, y)$ . В результате получается алгоритм художника на уровне пикселей. Здорово, правда? Конечно же, здесь имеется множество различных деталей, которые обязательно будут рассмотрены. Однако сначала поговорим о некоторых очевидных проблемах.

## Проблемы, связанные с Z-буфером

Во-первых, нужно определить, каким способом будет вычисляться значение координаты  $z$  для каждого многоугольника, для которого производится растеризация. Может оказаться, что это не так просто, поскольку к тому времени, когда дело доходит до растеризации, обычно вся информация о расположении многоугольников по оси  $z$  теряется. Эта проблема требует решения; необходимо также определить, в каком "пространстве" вычисляются значения  $z$ . Далее, предположим, что есть возможность извлечь информацию о координатах  $z$  в функции растеризации, но кто сказал, что мы хотим вычислять эти значения? Ведь это дополнительная работа для многострадальной функции. Она и так уже перегружена освещением, наложением текстуры, отсечением — мы просто загубим бедняжку. **Наконец**, следует учесть, что Z-буфер будет занимать такой же или больший объем памяти, чем буфер дисплея, поэтому объем необходимой памяти удваивается.

Поведем итог. Добавление Z-буфера приведет к замедлению работы функции растеризации, перерисовыванию пикселей и к тому, что потребуется, как минимум, в два раза больше памяти! Не похоже, чтобы это нам очень помогло. Однако интересно, что когда мы займемся наложением текстуры с учетом перспективы, то все равно будем вынуждены вычислять **значе-**

ния координат z, поэтому они в любом случае понадобятся нам в дальнейшем. Что касается проблемы, связанной с перерисовыванием, то для ее решения существуют другие пути. Однако если используется "чистый" подход с использованием Z-буфера, то придется с этим смириться. Поэтому лучший способ — сделать эту часть кода высокопроизводительной. Наконец, заметим, что современный компьютер в среднем имеет 128 Мбайт памяти, поэтому лишние 2 Мбайт, необходимые для Z-буфера размером 800×600, каждый элемент которого занимает 32 бита, — это не такая уж и большая проблема.

## Примеры Z-буфера

Перед тем как реализовать Z-буфер, убедимся, что мы правильно понимаем принципы его работы. Для этого рассмотрим небольшой пример. Предположим, что имеется Z-буфер размером 10×10 пикселей (для компьютеров, которыми пользовались в конце 70-х, этого вполне хватало!), и что каждый его элемент представлен 8-битовым целым числом. Если число имеет дробную часть, оно преобразуется в целое значение путем округления. Для хранения этого Z-буфера можно было бы воспользоваться структурой данных, представленной ниже

```
UCHAR zbuffer[10][10];
```

или, возможно, такой:

```
UCHAR zbuffer[10*10];
```

На самом деле, конкретный вид структуры не имеет значения, лишь бы она подходила для наших целей. Для определенности предположим, что доступ к массиву осуществляется с помощью выражения `zbuffer[xi][yi]`, где  $\{0 \leq xi \leq 9\}$  и  $\{0 \leq yi \leq 9\}$ . Чтобы сделать пример интереснее, предположим, что экран также имеет вид массива 10×10. Назовем его `screen_buffer[][]` и будем придерживаться той же схемы адресации, что и для Z-буфера.

Начнем с того, что инициализируем Z-буфер значениями, которые наверняка никогда не будут в него записаны в ходе растеризации.

```
zbuffer[][] =
```

```
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255
```

Все элементы массива `screen_buffer` инициализируем нулями, что в нашем примере будет соответствовать черному цвету.

```
screen_buffer[][] =
```

```
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
```

```
0000000000
0000000000
0000000000
0000000000
```

Теперь выведем прямоугольник (это проще, чем треугольник) с вершинами в точках (0,0) и (3,3). Пусть ему соответствует значение  $z=100$  и цвет с индексом 5. Поскольку 100 меньше, чем 255, то это значение будет записано вместо старого значения 255 в тех элементах Z-буфера, где расположен прямоугольник. Соответственно, значения пикселей экрана нужно будет заменить индексом цвета 5. Массивы `zbuffer[]` и `screen_buffer[]` теперь выглядят следующим образом.

`zbuffer[] =`

```
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
```

`screen_buffer[] =`

```
5555000000
5555000000
5555000000
5555000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
```

НА ЗАМЕТКУ

Измененные значения **массивов** `zbuffer[]` и `screen_buffer[]` выделены полужирным шрифтом.

Теперь можно продвигаться дальше. На этот раз давайте выведем прямоугольник, противоположные вершины которого расположены в точках (2,2) и (5,4). Пусть координата  $z$  этого прямоугольника равна 150, а цвет имеет индекс 8. 150 — это дальше, чем 100, но ближе, чем 255. В итоге получится следующее.

`zbuffer[] =`

```
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 150 150 255 255 255 255
100 100 100 100 150 150 255 255 255 255
255 255 150 150 150 150 255 255 255 255
255 255 255 255 255 255 255 255 255 255
```

```
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
```

```
screen_buffer[] []-
```

```
5555000000
5555000000
5555880000
5555880000
0088880000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
```

**НА ЗАМЕТКУ**

Измененные значения массивов `zbuffer[] []` и `screen_buffer[] []` выделены полужирным шрифтом.

Как видим, первый прямоугольник обладает в Z-буфере приоритетом перед новым многоугольником, поэтому область их перекрытия осталась неизменной. Однако оставшаяся часть, которая расположена правее и ниже, привела к обновлению массивов `zbuffer[] []` и `screen_buffer[] []`. Наконец, добавим еще один прямоугольник. На этот раз его противоположные вершины будут расположены в точках (0,0) и (9,9), координата z равна 50, а его цвет имеет индекс 1.

```
zbuffer[] [] =
```

```
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
```

```
screen_buffer[] []=
```

```
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

Как видим, все элементы Z-буфера и буфера дисплея изменили свои значения! Это самый нежелательный случай. Не хотелось бы, чтобы он возникал слишком часто, потому что получается, что пиксели многократно перерисовываются впустую. В защиту применения Z-буфера по сравнению с Z-сортировкой заметим, что при Z-сортировке делается то же самое, поскольку в ее ходе каждый многоугольник выводится полностью, независимо ни от каких обстоятельств.

НА ЗАМЕТКУ

По сравнению с Z-сортировкой, Z-буфер обладает одной подкупающей особенностью. Она проявляется в тех случаях, когда новый пиксель не перезаписывается поверх предыдущих значений, полученных при растеризации. При этом удается избежать его прорисовки. Различие оказывается существенным, когда наложение текстуры и моделирование освещения выполняется на уровне пикселей.

Для введения этого вполне достаточно. Теперь можно изучить вопрос более глубоко и попытаться понять, как именно будут вычисляться значения координат  $z$  для многоугольников и как будет осуществляться их растеризация.

## Метод уравнения плоскости

Итак, рассмотрим процесс вычисления значений координаты  $z$ , соответствующих каждому пикселю сканируемого многоугольника. Это можно сделать несколькими способами, однако мы рассмотрим два простейших. Первый метод заключается в использовании определения плоскости в процессе сканирования. Известно, что все вершины многоугольника находятся в одной плоскости. Это требование касается всех многоугольников, обрабатываемых нашей системой. Поэтому в процессе сканирования треугольника можно сначала найти уравнение плоскости, в которой лежит этот треугольник. Для этого вычислим координаты нормали к плоскости многоугольника, а затем запишем уравнение плоскости, заданное с помощью точки, через которую она проходит, и вектора нормали. Оно имеет вид

$$n_x(x - x_0) + n_y(y - y_0) + n_z(z - z_0) = 0,$$

где  $\langle n_x, n_y, n_z \rangle$  — координаты вектора нормали к сканируемому многоугольнику, а  $(x_0, y_0, z_0)$  — точка, принадлежащая многоугольнику.

Далее это уравнение можно преобразовать к виду

$$ax + by + cz + d = 0,$$

где  $a = n_x$ ,  $b = n_y$ ,  $c = n_z$ ,  $d = -(n_x x_0 + n_y y_0 + n_z z_0)$ .

Решая полученное уравнение относительно координаты  $z$ , получим:

$$z = -(ax + by + d)/c.$$

Это уравнение позволяет найти величину  $z$ , поскольку координаты  $(x, y)$  каждого пикселя известны, ведь они генерируются в процессе сканирования. Единственный недостаток описанного выше метода заключается в том, что в нем необходимо найти уравнение плоскости и определить константы, а затем выполнить два умножения, одно деление и два сложения. Безусловно, эти вычисления можно сократить. Во-первых, можно использовать предварительно вычисленные нормали к поверхностям, входящие в состав структуры, в которых хранятся параметры каждого многоугольника. Во-вторых, можно кардинально упростить вычисления координат  $z$  с помощью метода, аналогичного тому, который применялся при затенении по Гуро. Для начала заметим, что при переходе от одного пикселя к другому координата  $x$  изменяется ровно на 1.0. В следующих разделах описано, как это свойство можно использовать для упрощения вычислений.

## Координата z как функция x

Заметим, что если

$$z_i = -(ax + by + d)/c,$$

то

$$\begin{aligned} z_{i+1} &= -(a(x+1) + by + d)/c = \\ &= -(ax + a + by + d)/c = \\ &= z_i + (-a/c). \end{aligned}$$

Нетрудно заметить, что разность между двумя значениями координат z, соответствующая соседним пикселям по горизонтали, равна  $(-a/c)$ . Поэтому при сдвиге на один пиксель в пределах строки все сводится к изменению координаты z путем добавления постоянного коэффициента. Конечно же, все еще нужно вычислить значение z для начального пикселя каждой строки, однако это тоже можно сделать с помощью интерполяции, поскольку при переходе из одной строки на другую координата y изменяется на 1.0. Проведем аналогичные вычисления для координаты z при изменении y.

## Координата z как функция y

Заметим, что если

$$z_i = -(ax + by + d)/c,$$

то

$$\begin{aligned} z_{i+1} &= -(ax + b(y+1) + d)/c = \\ &= -(ax + by + b + d)/c = \\ &= z_i + (-b/c). \end{aligned}$$

Это означает, что при переходе на одну строку ниже новое значение координаты z равно старому значению плюс константа  $(-b/c)$ . Соотношение между приращениями координат по разным осям при сканировании многоугольника проиллюстрировано на рис. 11.5.

Несмотря на то, что непосредственное вычисление координат z многоугольника с помощью уравнения плоскости работает корректно, его лучше не применять по двум причинам: 1) этот метод правильно работает только в пространстве объекта (к этой концепции мы еще вернемся), 2) значения координат z так же легко вычисляются с помощью интерполяции. Аналогично интерполированию интенсивностей по трем вершинам треугольника с затенением по Гуро, можно интерполировать и величины координат z.

## Интерполяция координаты z

Обратите внимание на рис. 11.6, на котором изображена схема, на конкретном примере поясняющая процесс интерполяции координаты z. Нетрудно заметить, что верхняя сторона треугольника горизонтальна. Эти вычисления аналогичны вычислениям для треугольника с горизонтальной нижней стороной. Таким образом, достаточно реализовать функцию для треугольника с горизонтальной верхней стороной, а конечная реализация программы будет обрабатывать оба эти частных случая.

Интерполяционные уравнения для величин  $z_{left}$ ,  $z_{right}$ , и  $z_{middle}$  имеют такой вид:

$$z_{left} = ((y_3 - y) * z_1 + (y - y_1) * z_3) / (y_3 - y_1);$$

$$z_{right} = ((y_3 - y) * z_2 + (y - y_1) * z_3) / (y_3 - y_1);$$

$$z_{middle} = ((x_e - x) * z_{left} + (x - x_s) * z_{right}) / (x_e - x_s);$$

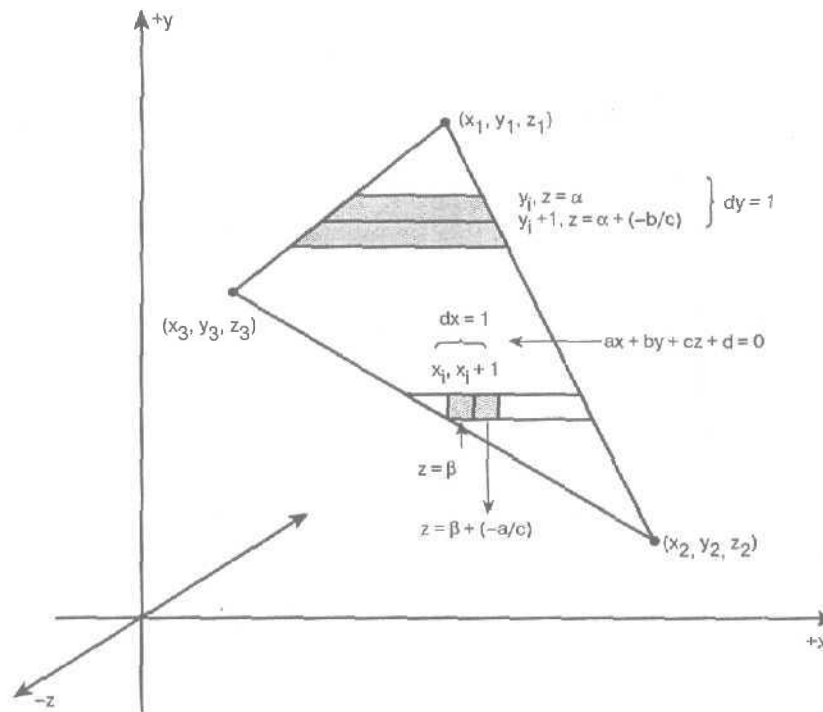


Рис. 11.5. Интерполяционные вычисления на плоскости

Эти уравнения, конечно, нельзя назвать простыми, потому что они записаны не в инкрементной форме, однако их легко преобразовать к нужному виду. Сначала найдем инкрементный вид уравнения для  $z_{\text{middle}}$  при изменении координаты  $x$ :

$$z_{\text{middle}_{i+1}} = ((x_e - x - 1) * z_{\text{left}} + (x + 1 - x_s) * z_{\text{right}}) / (x_e - x_s) \\ = ((x_e - x) * z_{\text{left}} - z_{\text{left}} + (x - x_s) * z_{\text{right}} + z_{\text{right}}) / (x_e - x_s)$$

$$= ((x_e - x) * z_{\text{left}} + (x - x_s) * z_{\text{right}} + z_{\text{right}} - z_{\text{left}}) / (x_e - x_s)$$

$$= ((x_e - x) * z_{\text{left}} + (x - x_s) * z_{\text{right}}) / (x_e - x_s) + (z_{\text{right}} - z_{\text{left}}) / (x_e - x_s)$$

$$= z_{\text{middle}_i} + (z_{\text{right}} - z_{\text{left}}) / (x_e - x_s)$$

Как видите, разность координат  $z$ , соответствующих двум соседним (расположенным в одной строке развертки) пикселям, равна  $(z_{\text{right}} - z_{\text{left}}) / (x_e - x_s)$ . Обозначив эту величину через  $\text{delta\_zx}$ , получим:

$$z_{\text{middle}_{i+1}} = z_{\text{middle}_i} + \text{delta\_zx}$$

Нам снова удалось сформулировать алгоритм, в котором очередное значение искомой величины определяется из предыдущего путем прибавления к нему константы. Однако для каждой строки развертки все еще нужно найти величины  $z_{\text{right}}$  и  $z_{\text{left}}$ . Для этого также можно воспользоваться методом разностей и разработать интерполяционный алгоритм, основанный на том факте, что соседним линиям развертки соответствуют коор-

динаты  $y$ , отличающиеся на 1.0. Опуская длинные преобразования, приведем конечный результат вычислений:

$$z\_left_{i+1} = z\_left_i + (z_3 - z_1) / (y_3 - y_1)$$

и

$$z\_right_{i+1} = z\_right_i + (z_3 - z_2) / (y_3 - y_1)$$

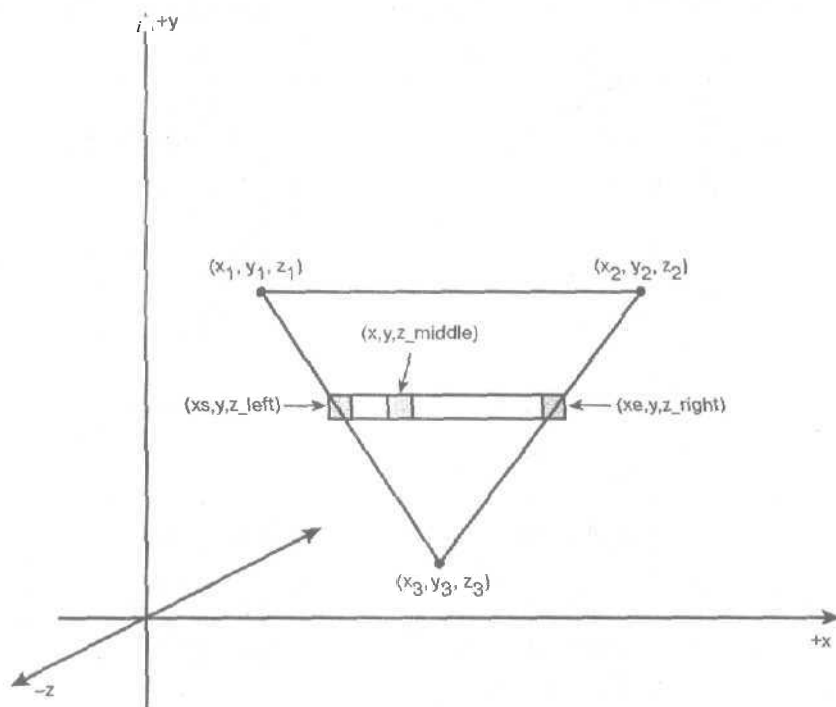


Рис. 11.6. Схема интерполяции координаты  $z$

Если ввести обозначения  $\delta_{zyl} = (z_3 - z_1) / (y_3 - y_1)$  и  $\delta_{zyr} = (z_3 - z_2) / (y_3 - y_1)$ , то эти формулы упростятся до

$$z\_left_{i+1} = z\_left_i + \delta_{zyl}$$

и

$$z\_right_{i+1} = z\_right_i + \delta_{zyr}$$

Согласно этим выражениям, значения  $z\_right$  и  $z\_left$  определяются из предыдущих (соответствующих предыдущей строке развертки) с помощью единственной операции сложения. Здорово, правда?

Теперь, когда мы получили интерполяционные соотношения, все, что нужно сделать, — это применить их вместе с физическим **Z-буфером** в новой функции, осуществляющей сканирование треугольника, — и дело в шляпе! Однако следует сделать пару замечаний. Во-первых, приведенный выше вывод **уравнений** — это, по сути, стандартная линейная интерполяция, поэтому новая функция не будет содержать ничего принципиально нового по сравнению с функциями наложения текстуры и затенения по Гуро. В обеих этих функциях уже реализована линейная интерполяция значений по **внутрен-**

ней области треугольника, поэтому достаточно добавить еще одну интерполируемую величину — значение координаты  $z$  — и все. Поэтому приведенный выше вывод уравнений нужен, если необходимо начать все сначала, но нам он ни к чему, поскольку у нас уже есть основа кода. Еще один не совсем очевидный факт заключается в том, что разработанный для **Z-буфера** алгоритм не совсем правильный!

## Проблемы, связанные с Z-буфером, и 1/Z-буферизация

Как уже было сказано, алгоритм работы Z-буфера в том виде, в котором он изложен в предыдущем разделе, неправильный. Причина заключается в том, что он разработан в мировой системе координат или в системе координат камеры. Мы пользуемся координатами треугольников  $(x, y, z)$  и забываем о том, что треугольники уже спроецированы, поэтому следует работать с проекциями  $(x', y')$  координат на пространство экрана! В этом и заключается основная проблема. Взглянув на рис. 11.7, вы поймете, о чем идет речь.

Оказывается, в трехмерном пространстве осуществлять интерполяцию по поверхности треугольника не очень удобно, поскольку нужно выводить двумерные треугольники на основе проекционных уравнений:

$$x_{\text{screen}} = d \cdot x / z$$

$$y_{\text{screen}} = d \cdot y / z$$

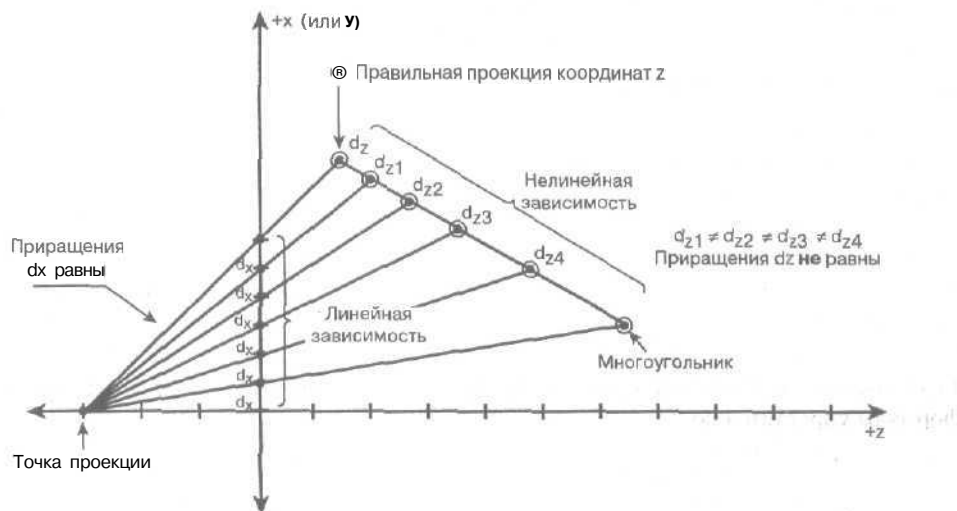


Рис. 11.7. Интерполяция координат  $z$  после проецирования в пространство экрана оказывается неверной

Взглянув на рис. 11.7, нетрудно заметить, что для интерполяции значений  $z$  исходного треугольника используются координаты  $x_{\text{screen}}$  и  $y_{\text{screen}}$ , полученные после проецирования на плоскость экрана. В результате **величины** координат  $z$  в пространстве экрана нелинейны, из-за чего происходит искажение данных, которые заносятся в **Z-буфер**. Аналогичный эффект имеет место при затенении по **Гуро** и аффинном отображении текстуры. Однако при затенении по **Гуро** и аффинном отображении текстуры искажения не так уж сильны, и алгоритм можно оставить в неизменном виде, особенно, если многоугольники достаточно малы и расположены параллельно плоскости наблюдения. То же

самое можно сказать по поводу использования в **Z-буфере** значений  $z$ , полученных путем интерполяции на основе координат  $(x, y)$  трехмерного треугольника. Вычисленные значения  $z$  окажутся неправильными, что хорошо видно из рис. 11.7. На этом рисунке лучи проведены так, что точки их пересечения с плоскостью экрана расположены с одинаковым шагом по оси  $x$ . Однако если проследить ход этих лучей до исходного неспроецированного многоугольника, то окажется, что координаты  $z$  соответствующих точек пересечения расположены нелинейно. Фактически расстояния между ними меняются по закону  $1/z$ . Математические выкладки, демонстрирующие эту закономерность подробнее, приводятся ниже, на этапе реализации отображения текстуры, правильно учитывающего перспективу. Пока же достаточно усвоить, что линейно в пространстве экрана изменяется не координата  $z$ , а величина  $1/z$ .

Зачем же тогда интерполировать величину  $z$ ? Почему бы не реализовать интерполяцию величины  $1/z$ , которая линейно изменяется в пространстве экрана? Хорошая идея, именно этим мы и займемся, однако это будет несколько позже. А пока что удовлетворимся линейной интерполяцией по  $z$  и забудем об искажении. Как оказывается, и при этом **Z-буфер** все равно будет работать правильно. Тем не менее, когда мы перейдем к корректному с точки зрения перспективы отображению текстуры, то переключимся на  $1/z$ -буферизацию. При этом для вычисления координат отображаемой текстуры используются значения  $1/z$ , и одним выстрелом будут убиты два зайца! Пока что не будем ничего усложнять, добавив **Z-буфер** в один программный модуль.

## Пример интерполяции по **Z** и $1/z$

Теперь, когда стала понятна причина неточностей, связанных со стандартным **Z-буфером**, и преимущества  $1/z$ -буфера, рассмотрим конкретный пример. Универсальный совет — чтобы знать что-либо наверняка, надо глубоко понять явления, лежащие в основе этих знаний.

Я всегда придерживался этой философии. Для этого я описываю алгоритм, черчу иллюстрирующие графики и блок-схему, тестирую, кодирую, еще что-нибудь придумываю и наконец убеждаюсь, что я действительно понимаю суть **вопроса**. Именно так приходит чувство уверенности в своих знаниях. В противном случае можно иметь общее представление о работе **какого-то** метода, но оно может оказаться неполным или неточным. Итак, рассмотрим простой пример, чтобы понять, что происходит при **Z-буферизации** и  $1/z$ -буферизации.

На рис. 11.8 приведена типичная схема аксонометрической проекции. Вертикальная ось — это ось  $y$ , горизонтальная — ось  $z$ , а ось  $x$  перпендикулярна плоскости рисунка и направлена на зрителя. Расстояние от точки наблюдения до плоскости, на которую осуществляется проецирование, равно двум единицам, и, кроме того, для простоты мы предположим, что координаты  $x$  всех точек равны нулю; другими словами, данный многоугольник находится в плоскости  $x = 0$ .

Для начала рассмотрим треугольник, который нужно спроецировать, растеризовать и занести в **Z-буфер**. Сначала обработаем сторону треугольника, соединяющую вершины  $p_0(0, 12, 4) \rightarrow p_1(0, 5, 7)$ . В результате проецирования на плоскость наблюдения получим вершины  $p'_0(0, y'_0, 0) \rightarrow p'_1(0, y'_1, 0)$ . Конечно же, нужно вычислить значения  $y'_0$  и  $y'_1$ , однако известно, что компоненты  $x$  и  $z$  равны 0. Вот здесь-то и начинается интересная часть: мы собираемся вычислить координаты спроецированных на экран точек  $p'_0$  и  $p'_1$ , а затем на их основе

выполнить интерполяцию, чтобы найти точки  $p'_1$  и  $p'_2$ , а также их прообразы  $p^*$  и  $p_b^*$ . Вскоре мы убедимся, что результат получится **неправильный**. Начнем с того, что протабулируем известные величины:

$$p_0 = (0, 12, 4), p_1 = (0, 5, 7)$$

Затем найдем координаты аксонометрических проекций с помощью формулы  $y' = (d/(d+z)) \cdot y$ . То же самое нужно выполнить и для координат  $x$ , однако в данном примере мы считаем, что  $x = 0$ :

$$p_0' = (0, (2/(2+4)) \cdot 12, 0) = (0, 4, 0)$$

$$p_1' = (0, (2/(2+7)) \cdot 5, 0) = (0, 1.111, 0)$$

Однако поскольку мы собирались осуществить линейную интерполяцию величин  $z$  на основе этих спроецированных точек, присвоим их  **$z$ -значениям** величины координат  $z$  прообразов точек  $p'_0$  и  $p'_1$ :

$$p_0' = (0, 4, 4)$$

$$p_1' = (0, 1.111, 7)$$

Теперь все готово к интерполяции. Вычислим **величины**  $dy$  и  $dz$ :

$$dy = p_1'(y) - p_0'(y) = (1.11 - 4.0) = -2.89$$

$$dz = p_1'(z) - p_0'(z) = (7 - 4) = 3$$

Для того чтобы выполнить растеризацию отрезка  $p'_0 \rightarrow p'_1$  в пространстве экрана, нужно просто выводить по одному пиксели, расположенные между его краями. Каждый раз, опускаясь на одну строку развертки вниз, мы уменьшаем значение координаты  $y$  на 1.0. Координату  $z$  можно было бы интерполировать с помощью величины  $dz/dy$ , т.е. величины, на которую изменяется **координата  $z$**  с изменением  $y$ :

$$dz/dy = (3)/(-2.89) = -1.03$$

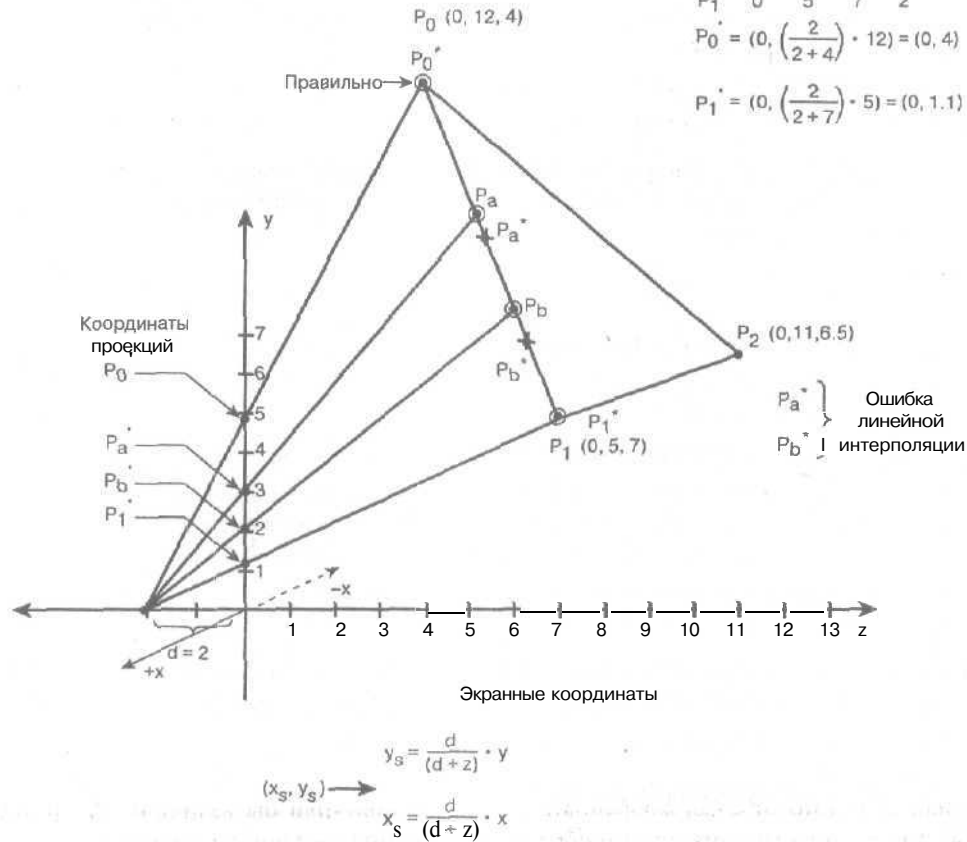
Это и есть то магическое число, которое свидетельствует об ошибочности линейной интерполяции. Мы начинаем с точки  $p'_0(0, 4)$  в пространстве экрана и выводим три пикселя. Описанные действия проиллюстрированы в табл. 11.1.

**Таблица 11.1. Интерполяция величины  $z$ ; пример 1**

Итерация	$x$	$y$	$z$
[0] Начало	$x = 0$	$y = 4$	$z = 4$
[1] $dy = -1, dz/dy = -1.03$	$x = 0$	$y = 3$	$z = 4 + 1.03 = 5.03$
[2] $dy = -1, dz/dy = -1.03$	$x = 0$	$y = 2$	$z = 5.03 + 1.03 = 6.06$
[3] $dy = -1, dz/dy = -1.03$	$x = 0$	$y = 1$	$z = 6.06 + 1.03 = 7.09$

Не важно, что мы зашли немного дальше, чем точка  $y=1.1$ . Напомним, что в реальной функции растеризации процесс был бы остановлен благодаря соглашению о **заполнении**, однако я захотел завершить этот шаг, чтобы у нас было достаточное количество примеров. Как видно из рис. 11.8, лучи, проведенные из точек  $p_a$  и  $p_b$  в точки  $p'_a$  и  $p'_b$ , представляют корректные интерполируемые величины для аксонометрической проекции. Это легко увидеть, если начертить схему на миллиметровке. Однако полученные в результате интерполяции точки  $p^*$  и  $p_b^*$  оказались смещенными! Причина этого **смеще-**

	x	y	z	d
$P_0$	0	12	4	2
$P_1$	0	5	7	2
$P_0'$	$= (0, (\frac{2}{2+4}) \cdot 12) = (0, 4)$			
$P_1'$	$= (0, (\frac{2}{2+7}) \cdot 5) = (0, 1.1)$			



Проблема решается очень просто. Поскольку от экранных координат линейно зависит не координата  $z$ , а обратная величина  $1/z$ , то и проводить интерполяцию следует именно для этой величины. Тогда в роли шага будет выступать не константа  $dz/dy$  (которая в нашем случае равна -1.03), а константа  $dy/dz$ .

$$\begin{aligned} \mathbf{p0}' &= (0, 4, 4) \\ \mathbf{p1}' &= (0, 1.11, 7) \end{aligned}$$

Заменяя значение координаты  $z$  величиной  $1/z$ , получим:

$$p_0' = (0, 4, 0.25)$$

$$p_1' = (0, 1.11, 0.142)$$

Теперь все готово для интерполяции. Вычислим величины  $dy$  и  $dz$  (конечно, вместо значений  $z$  мы воспользуемся значениями  $1/z$ ):

$$dy = p_1'(y) - p_0'(y) = (1.11 - 4.0) = -2.89$$

$$dz = p_1'(z) - p_0'(z) = (1/7 - 1/4) = -0.107$$

Чтобы выполнить растеризацию отрезка  $p_0' \rightarrow p_1'$  в экранном пространстве, как и в предыдущем случае, следует по одному выводить пиксели, расположенные между концами этого отрезка. Каждый раз номер строки развертки уменьшается на единицу, что соответствует уменьшению координаты  $y$  на величину 1.0. Координату  $z$  можно было бы интерполировать с помощью величины  $dz/dy$ , которая равна изменению координаты  $z$  при изменении координаты  $y$  на 1:

$$dz/dy = (-0.107)/(-2.89) = 0.037$$

Это число вновь иллюстрирует несостоятельность линейной интерполяции. Выведем три пикселя, начиная с точки  $p_0'(0, 4)$  в экранном пространстве. Все действия шаг за шагом продемонстрированы в табл. 11.2.

**Таблица 11.2. Интерполяция величины  $z$ ; пример 2**

Итерация	$x$	$y$	$z$
[0] Начало	$x = 0$	$y = 4$	$z = 1/4 = 0.25$
[1] $dy = -1, dz/dy = -0.037$	$x = 0$	$y = 3$	$z = 0.25 - 0.037 = 0.213$
[2] $dy = -1, dz/dy = -0.037$	$x = 0$	$y = 2$	$z = 0.213 - 0.037 = 0.176$
[3] $dy = -1, dz/dy = -0.037$	$x = 0$	$y = 1$	$z = 0.176 + 0.037 = 0.139$

Результаты получились на удивление правильными, несмотря на то, что последнее значение несколько меньше истинного! Это произошло потому, что при растеризации мы зашли немного дальше, чем следовало бы. Если бы мы растеризовали не 3.0 пикселя, а ровно 2.89 (это отвечает координате  $y=1.1$ ), то получили бы величину  $1/z$ , равную 0.142, что в точности соответствует обратному значению координаты  $z$  точки  $p_1'$ .

## Создание системы Z-буферизации

Но достаточно теории. Займемся лучше реализацией Z-буфера. Для этого нам надо зарезервировать некоторую область памяти, возможно, какие-то флаги или атрибуты, с помощью которых можно было бы задавать определенные его характеристики, и, пожалуй, этого будет достаточно. Ниже представлена первая версия.

```
// Директивы define для Z-буфера.
#define ZBUFFER_ATTR_16BIT 16
#define ZBUFFER_ATTR_32BIT 32
```

```
// Структура Z-буфера
typedef struct ZBUFFERV1_TYP
{
```

```

int attr;    // Атрибуты Z-буфера
UCHAR *zbuffer; // Указатель на хранилище
int width;   // Ширина в пикселях
int height;  // Высота в пикселях
int sizeq;   // Полный размер Z-буфера в
              // количестве величин типа QUAD
} ZBUFFERV1, *ZBUFFERV1_PTR;

```

В этом типе выделяется память для хранения Z-буфера, содержится информация о его размере в "словах", а также полный размер в 32-битовых значениях типа QUAD. Поля width и height представляют общую ширину и высоту Z-буфера, а поле sizeq — количество содержащихся в нем значений типа QUAD (каждое такое значение занимает четыре байта). Кроме того, имеется поле атрибута attr. На данном этапе оно просто отслеживает, является ли Z-буфер 16-битовым или 32-битовым. Я предусмотрел возможность использования 32-битового Z-буфера, хотя пока что он нам не понадобится; но позже, на этапе оптимизации, возможно, мы воспользуемся этим атрибутом. С Z-буфером необходимо иметь возможность производить такие действия:

- создавать;
- удалять;
- очищать Z-буфер или заполнять его какими-то значениями (причем делать это быстро).

Приведем код функции, которая **создает Z-буфер**.

```

int Create_Zbuffer(
    ZBUFFERV1_PTR zb, // Указатель на объект Z-буфера
    int width,        // Ширина
    int height,       // Высота
    int attr)         // Атрибуты Z-буфера
{
    // Эта функция создает Z-буфер заданной ширины, высоты
    // и с указанной в байтах длиной слов

    // Проверка корректности объекта
    if (!zb)
        return (0);

    // Проверка наличия выделенной ранее памяти
    if (zb->zbuffer)
        free(zb->zbuffer);

    // Присвоение полям указанных значений
    zb->width = width;
    zb->height = height;
    zb->attr = attr;

    // Проверка, является Z-буфер 16- или 32-битовым
    if (attr & ZBUFFER_ATTR_16BIT)
    {
        // Вычисление размера, выраженного в количестве
        // значений типа quad
        zb->sizeq = width*height/2;
    }
}

```

```

// Выделение памяти
if ((zb->zbuffer = (UCHAR *)malloc(width * height *
                                   sizeof(SHORT))))
    return(1);
else
    return (0);

} // if
else if (attr & ZBUFFER_ATTR_32BIT)
{
    // Вычисление размера, выраженного в количестве
    // значений типа quad
    zb->sizeq=width*height;

    // Выделение памяти.
    if ((zb->zbuffer = (UCHAR *)malloc(width * height *
                                       sizeof(INT))))
        return(1);
    else
        return (0);
} // if
else
    return (0);

} // Create_Zbuffer

```

В эту функцию передается указатель на создаваемый объект ZBUFFERV1, а также его ширина, высота и атрибуты. Например, чтобы создать 32-битовый Z-буфер 800x600, следует вызвать данную функцию со следующими параметрами.

```

ZBUFFERV1 zbuffer; //Хранилище создаваемого Z-буфера
Create_Zbuffer(&zbuffer, 800, 600, ZBUFFER_ATTR_32BIT);

```

Здесь есть один интересный момент: на самом деле хранилище Z-буфера не имеет типа. Ему присвоен тип UCHAR, поэтому его можно преобразовать к любому типу, какому мы только пожелаем — int, float и т.д. В функции растеризации, которую мы собираемся написать, Z-буфер будет преобразован к типу unsigned int, однако в своих функциях вы можете выбрать любой другой тип.

Далее нам понадобится функция, которая очищает Z-буфер или заполняет его каким-то значением. Как и создание буфера кадров, эту операцию нужно выполнять для каждого кадра. Обычно Z-буфер заполняется максимально возможными значениями для данного численного интервала или бесконечно большими значениями. Приведем код соответствующей функции.

```

void Clear_Zbuffer(ZBUFFERV1_PTR zb, UINT data)
{
    //Эта функция очищает Z-буфер и заполняет его
    // значениями. Заполнение ВСЕГДА производится на основе
    // величин типа QUAD. Поэтому если Z-буфер 16-битовый,
    // значения типа QUAD необходимо составить из двух
    // 16-битовых величин. В противном случае в функцию для
    // заполнения Z-буфера передается значение,
    // преобразованное к типу UINT
}

```

```
Mem_Set_QUAD((void *)zb->zbuffer, data, zb->sizeq);
} // Clear_Zbuffer
```

Нетрудно заметить, что в этой функции вызывается функция `Mem_Set_QUAD()`, которая написана на встроенном ассемблере и находится в библиотеке `T3DLIB1.H`.

```
inline void Mem_Set_QUAD(void *dest UINT data, int count)
{
    // Эта функция заполняет память беззнаковыми
    // 32-битовыми значениями
    // count - это количество значений типа quad

    asm
    {
        mov edi, dest ; Переменная edi указывает на целевую
                        ; ячейку памяти
        mov ecx, count ; Количество перемещаемых 32-битовых
                        ; слов
        mov eax, data ; 32-битовые данные
        rep stosd ; Перемещение данных
    } // asm
} // Mem_Set_QUAD
```

Встроенный ассемблер используется для заполнения Z-буфера с максимальной скоростью (на самом деле еще большей производительности можно было бы добиться с помощью **SIMD-команд**). Вызывая эту функцию для очистки Z-буфера, данные, конечно же, необходимо привести к типу `unsigned int`. Кроме того, если Z-буфер 16-битовый, должны быть созданы и соединены в одно целое два слова с максимальными значениями (рис. 11.9).

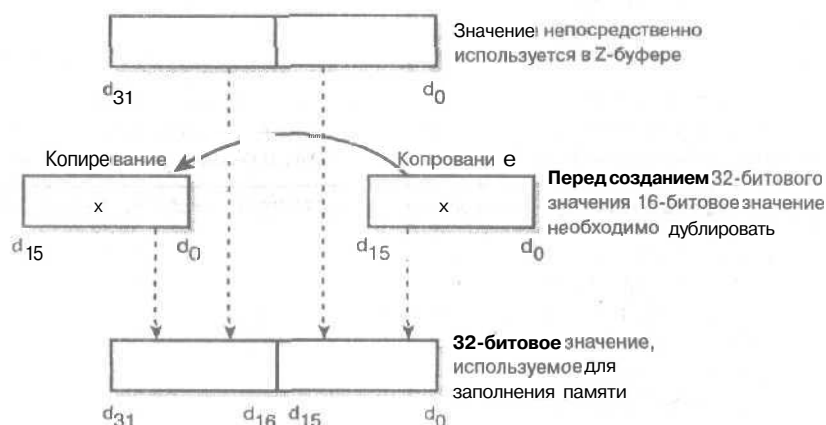


Рис. 11.9. Подготовка к скоростному заполнению Z-буфера значениями длиной в 32-бита

Например, если используется 16-битовый Z-буфер, в него нельзя просто так передать 32-битовое значение, составленное из двух максимальных 16-битовых. Сначала такое 32-битовое значение следует сконструировать, чтобы можно было заполнять память величинами типа `QUAD`. Алгоритм заполнения смежных ячеек выглядит следующим образом.

```
USHORT zmax16 - 10000; // Пусть максимальное значение
                        // координаты z равно 10000.
```

```
unsigned int zdata =
    (((unsigned int)zmax16) << 16 | (unsigned int)zmax16);
```

В качестве еще одного примера рассмотрим, как каждой ячейке 32-битового Z-буфера присваивается 32-битовое значение с фиксированной точкой, равное 32000.

```
Clear_Zbuffer(&zbuffer,(32000 << FIXP16_SHIFT));
```

Наконец, чтобы удалить Z-буфер и освободить память, нужно воспользоваться функцией `int Delete_Zbuffer(ZBUFFERV1_PTR zb)`

```
{
    // Эта функция удаляет Z-буфер и освобождает
    // выделенную для него память

    // Проверяем корректность объекта
    if (zb)
    {
        // Удаляем память и обнуляем объект
        if (zb->zbuffer)
            free(zb->zbuffer);

        // Очищаем память.
        memset((void *)zb,0, sizeof(ZBUFFERV1));

        return(1);
    } // if
    else
        return(0);
} // Delete_Zbuffer
```

Достаточно вызвать функцию `Delete_Zbuffer()`, передав в нее указатель на объект Z-буфера, который нужно удалить, — и функция освободит память и удалит объект.

**СОВЕТ**

Для Z-буфера создан абстрактный тип данных `ZBUFFERV1`, чтобы в дальнейшем можно было оперировать с несколькими Z-буферами и попытаться создать определенные спецэффекты.

## Добавление поддержки Z-буфера в функцию растеризации

Прежде чем добавлять поддержку Z-буфера в функции растеризации, нужно определить, какие функции должны иметь такую поддержку. В настоящий момент у нас есть такие функции растеризации:

- класс 1: функции для плоского и постоянного затенения;
- класс 2: функции для затенения по Гуро;
- класс 3: функции для наложения текстуры с постоянным затенением;
- класс 4: функции для наложения текстуры с плоским затенением.

Самыми старыми из всех функций растеризации являются те, что **принадлежат** к первому классу. Они остались **еще** от программ, которые создавались для предыдущей книги *Программирование игр для Windows. Советы профессионала*, только работают не с **переменными** с фиксированной точкой, а со смешанными, т.е. и с целыми числами, и с числами с фиксированной точкой. Нельзя сказать, чтобы эти функции работали медленнее или быстрее, чем функции классов 2–4, в основе которых лежит один и тот же принцип растеризации и интерполяции, — просто функции класса 1 другие. Что же касается функций классов 2–4, то в одном случае в них интерполируются цвета (для затенения по Гуро), а в других — координаты текстуры (для отображения текстуры). Какие же функции растеризации следует обновить? Неплохо было бы создать новые версии всех этих функций! Однако сначала займемся функциями класса 1, в которых осуществляется растеризация многоугольников с плоским и постоянным затенением, поскольку в них не производится интерполяция, и внедрить Z-буфер в них труднее, чем в другие. Таким образом, все функции этого класса нужно будет переделать.

А теперь о хорошем: для обновления всех остальных функций в каждую из них достаточно добавить всего по 10 строк кода! Возможно, несколько **больше**, но не намного, потому что во всех функциях уже реализована интерполяция по двум или более переменным. Это может быть интерполяция цветов в формате RGB в функции затенения по Гуро или интерполяция координат текстуры в функции для отображения текстуры. Нужно просто добавить координаты z каждого многоугольника, выполнить их интерполяцию и закодировать во внутренних циклах логику работы Z-буфера. Перепишем все функции растеризации одну за другой.

### Обновление функции для плоского затенения

Труднее всего создавать новую версию функции, предназначенной для плоского затенения, потому что ее придется полностью переделать. В качестве основы будут взяты фрагменты кода из функции, осуществляющей затенение по Гуро, в которой производится интерполяция цветов в формате RGB. Мы используем один канал цвета и заменим его координатой z каждой вершины. Для этого, во-первых, нужно убедиться, что в новую функцию (как бы мы ее не назвали) передается структура, содержащая параметры каждого многоугольника, в том числе координаты всех его вершин. Напомним, что при вызове функций растеризации этого класса координаты z в них не передаются. Взгляните на функцию `Draw_RENDERLIST4DV2_Solid16()`, код которой скопирован из библиотеки T3DLIB7.CPP и приведен ниже.

```
void Draw_RENDERLIST4DV2_Solid16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer,
    int lpitch)
{
    if (!(rend_list->poly_ptrs[poly]->state &
        POLY4DV2_STATE_ACTIVE) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV2_STATE_CLIPPED) ||
        (rend_list->poly_ptrs[poly]->state &
        POLY4DV2_STATE_BACKFACE) )
        continue; // Переходим к следующему многоугольнику

    // Сначала нужно определить, обладает ли многоугольник
    // текстурой, потому что многоугольники с текстурой могут
    // быть с постоянным или плоским затенением, а в
```

```

// зависимости от типа затенения вызываются те или иные
// функции растеризации
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
    // Задаем параметры вершины
    face.tvlist[0].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].u0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].u0;
    face.tvlist[0].v0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].v0;

    face.tvlist[1].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].u0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].u0;
    face.tvlist[1].v0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].v0;

    face.tvlist[2].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].u0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].u0;
    face.tvlist[2].v0 =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].v0;

    // Назначаем текстуру
    face.texture = rend_list->poly_ptrs[poly]->texture;

    // Это текстура с постоянным затенением?
    if (rend_list->poly_ptrs[poly]->attr &
        POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
    {
        // Выводим треугольник с текстурой
        // и постоянным затенением
        Draw_Textured_Triangle16(&face, video_buffer, lpitch);
    } // if
    else
    {
        // Выводим треугольник с плоским затенением
        face.lit_color[0] = rend_list->poly_ptrs[poly]->
            lit_color[0];
        Draw_Textured_TriangleFS16(&face, video_buffer,
            lpitch);
    } // else
}

```

```

    } // if
else
if ((rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
    (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
{
    // Выводим треугольник с помощью основной функции,
    // осуществляющей плоское затенение
    Draw_Triangle_2D2_16(
        rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->lit_color[0],
        video_buffer, lpitch);
} // if
else
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
{
    // {Далее будут использованы преимущества
    // этих структур данных..}
    // Задаем параметры вершин
    face.tvlist[0].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.lit_color[0] =
        rend_list->poly_ptrs[poly]->lit_color[0];

    face.tvlist[1].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.lit_color[1] =
        rend_list->poly_ptrs[poly]->lit_color[1];

    face.tvlist[2].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.lit_color[2] =
        rend_list->poly_ptrs[poly]->lit_color[2];

    // Выводим треугольник с затенением по Гуро
    Draw_Gouraud_Triangle16(&face, video_buffer, lpitch);
} // if gouraud
} // for poly
} // Draw_RENDERLIST4DV2_Solid16

```

Приведенная выше функция содержит три основные выполняемые ветви: для многоугольников с постоянным и плоским затенением, для многоугольников с затенением по Гуро и для многоугольников с текстурой (необязательно в указанном порядке). В каждой ветви происходит вызов одной из следующих функций:

- `Draw_Gouraud_Triangle16()` — стандартная функция, **осуществляющая** затенение по Гуро;
- `Draw_Textured_Triangle16()` — функция для отображения текстуры с постоянным затенением;
- `Draw_Textured_TriangleFS16()` — функция для отображения текстуры с плоским затенением;
- `Draw_Triangle_2D2_16()` — старая функция растеризации многоугольников с плоским и постоянным затенением.

Мы начали переделывать последнюю функцию, однако, по сути, нужно переписать их все, включая саму функцию `Draw_RENDERLIST4DV2_Solid16()`. Ниже представлен прототип новой функции для растеризации многоугольников с плоским и постоянным затенением.

```
void Draw_Triangle_2DZB_16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR * _dest_buffer, // Указатель на видеобuffer
    int mem_pitch, // Количество байтов в строке
    UCHAR * _zbuffer, // Указатель на Z-буфер
    int zpitch) // Количество байтов в строке
    // Z-буфера
```

Этот прототип отличается от старого. В предыдущей версии не было параметра `POLYF4DV2_PTR`; в нее явным образом передавались параметры вершин и цвет многоугольника. Однако теперь мы нуждаемся в унификации вызываемых процедур, поэтому с настоящего момента все функции растеризации будут обладать одним и тем же набором параметров. Еще одно новшество состоит в том, что добавлены указатели на массив, в котором хранится Z-буфер, и переменную, содержащую выраженную в байтах длину строки Z-буфера. Ответственность за организацию корректного доступа к Z-буферу возлагается на функцию растеризации, а за выделение для Z-буфера достаточного количества памяти — на вызывающую функцию. Таким образом, чтобы вызвать функцию `Draw_Triangle_2DZB_16()`, нужно выполнить примерно следующие действия.

```
face.lit_color[0] =
    rend_list->poly_ptrs[poly]->lit_color[0];

// Задаем параметры вершин
face.tvlist[0].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
face.tvlist[0].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].z;

face.tvlist[1].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].z;
```

```

face.tvlist[2].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].z;

// Выводим треугольник с помощью основной функции,
// осуществляющей растеризацию многоугольников с
// плоским затенением
Draw_Triangle_2DZB_16(&face, video_buffer, lpitch,
    zbuffer, zpitch);

```

В приведенном примере параметры вершин многоугольника, содержащегося в списке визуализации, копируются в переменную, в которой хранятся параметры поверхности, после чего указатель на эту поверхность передается в функцию `Draw_Triangle_2DZB_16()`. Переменной цвета присваивается значение `lit_color[0]`, поскольку для многоугольников с плоским и постоянным затенением нужно задавать только этот цвет. Наконец, вызывается функция растеризации, и в нее передаются указатели на видеобuffer `video_buffer` и его шаг `lpitch`, а также указатель на Z-буфер `zbuffer` и его шаг `zpitch`. Все это должно быть вам знакомо — именно так работают все остальные функции растеризации с тех пор, как были **разработаны** модули освещения и отображения текстуры.

Вот почти и все, что следует сказать по поводу вызова функции `Draw_Triangle_2DZB_16()` и ее применения. Листинг этой функции привести нет возможности в силу большого объема. Поэтому ниже представлены только некоторые фрагменты, благодаря которым можно понять принцип работы кода, **реализующего Z-буфер**.

```

// Присваиваем указателю на экран значение,
// соответствующее началу строки
screen_ptr = dest_buffer + (ystart * mem_pitch);

// Присваиваем указателю на Z-буфер значение,
// соответствующее началу строки
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi <= yend; yi++)
{
    // Определяем промежуток между конечными точками
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Определяем начальные значения точек u, v, w,
    // используемых для интерполяции
    zi = zl + FIXP16_ROUND_UP;

    // Вычисляем интерполируемые величины u, v
    if ((dx - (xend - xstart)) > 0)
    {
        dz = (zr - zl) / dx;
    } // if
    else
    {

```

```

    dz = (zr - zl);
} // else

// Выводим строку развертки
for (xi=xstart; xi<=xend; xi++)
i
    // Проверяем, меньше ли координата z текущего
    // пикселя, чем значение, которое хранится в
    // Z-буфере
    if (zi < z_ptr[xi])
    {
        // Выводим пиксель текстуры в формате 5.6.5
        screen_ptr[xi] = color;

        // Обновляем Z-буфер.
        z_ptr[xi] = zi;
    } // if

    // Интерполируем координату z
    zi+=dz;
} // for xi

// Интерполируем координаты x и z вдоль
// правой и левой сторон треугольника
xl+=dxndl;
zl+=dzndl;
xr+=dxndr;
zr+=dzndr;

// Смещаем указатель на экран
screen_ptr+=mem_pitch;

// Смещаем указатель на Z-буфер
z_ptr+=zpitch;
} // for yi

```

Приведенный выше код выводит одну за другой строки развертки для каждого значения  $y$ . Обратите внимание на код, **обрабатывающий Z-буфер**, — он выделен полужирным шрифтом, и в нем реализованы обсуждавшиеся выше действия. Сначала текущее значение координаты  $z$  сравнивается с тем, которое хранится в Z-буфере.

```

if (zi < z_ptr[xi])
{

```

Если проверка проходит успешно, выполняется растеризация пикселя.

```

// Вывод пикселя текстуры в формате 5.6.5
screen_ptr[xi] = color;

```

Далее происходит обновление Z-буфера.

```

// Обновляем Z-буфер
z_ptr[xi] = zi;
} // if

```

Заметим, что после того, как установлено, что хранящееся в Z-буфере значение нужно обновить, оно не обновляется сразу же значением  $z_i$ . Это трюк, связанный с оптимизацией. Обычно не следует производить запись значения сразу же после его считывания; лучше выполнить какие-нибудь промежуточные действия, и лишь потом — запись.

В приведенном выше фрагменте — суть функции, предназначенной для реализации плоского и постоянного затенения. Все другие части остались неизменными; просто был добавлен код для **интерполяции** координаты  $z$  по области **треугольника**, а затем производилось сравнение **величин**  $z$  и обновление Z-буфера, если это необходимо.

## Обновление функции затенения по Гуро

Добавить поддержку Z-буфера в функцию, **осуществляющую** затенение по Гуро, было легко, поскольку в ней уже реализована интерполяция по трем каналам цвета в формате RGB. Все свелось к добавлению еще одной интерполируемой величины. Ниже приводится прототип новой функции растеризации. Он отличается только именем и наличием параметров, соответствующих Z-буферу и его шагу.

```
void Draw_Gouraud_TriangleZB16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR * _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,        // Количество байтов в
                        // строке: 320, 640 и т.д.
    UCHAR * zbuffer,      // Указатель на Z-буфер
    int zpitch)           // Количество байтов в строке
                        // Z-буфера
```

Вызов функции производится точно так же, как и до этого, только теперь в нее нужно передавать дополнительные параметры: указатель на хранилище Z-буфера и заданную в байтах длину его строки (эти параметры имеют тот же смысл, что и в функции для постоянного и плоского затенения). А теперь изучим функцию, реализующую затенение по Гуро, и посмотрим, как выглядит ее внутренний цикл, поддерживающий Z-буфер.

```
screen_ptr = dest_buffer + (ystart * mem_pitch);
```

```
// Устанавливаем указатель на Z-буфер в начало строки
z_ptr = zbuffer + (ystart * zpitch);
```

```
for (yi = ystart; yi <= yend; yi++)
```

```
{
```

```
    // Вычисляем расстояние между конечными точками
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend   = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
```

```
    // Вычисляем начальные значения интерполируемых
```

```
    // величин u, v, w
```

```
    ui = ul + FIXP16_ROUND_UP;
```

```
    vi = vl + FIXP16_ROUND_UP;
```

```
    wi = wl + FIXP16_ROUND_UP;
```

```
    zi = zl + FIXP16_ROUND_UP;
```

```
    // Вычисляем интерполируемые величины u, v
```

```
    if ((dx - (xend - xstart)) > 0)
```

```
{
```

```
    du = (ur - ul) / dx;
```

```
    dv = (vr - vl) / dx;
```

```

    dw=(wr-wl)/dx;
    dz=(zr-zl)/dx;
} // if
else
{
    du=(ur-ul);
    dv=(vr-vl);
    dw=(wr-wl);
    dz=(zr-zl);
} // else

////////////////////////////////////

// Проверяем, нужно ли выполнять отсечение по левому
// краю
if (xstart < min_clip_x)
{
    // Определяем перекрытие по x
    dx = min_clip_x - xstart;

    // Даем приращение интерполируемым величинам
    ui+=dx*du;
    vi+=dx*dv;
    wi+=dx*dw;
    zi+=dx*dz;

    // Обновляем переменные
    xstart = min_clip_x;
} // if

// Проверяем, нужно ли выполнять отсечение по правому
// краю
if (xend > max_clip_x)
    xend = max_clip_x;

////////////////////////////////////

// Выводим строку развертки
for (xi=xstart; xi<=xend; xi++)
{
    // Проверяем, меньше ли текущая координата z того
    // значения, которое хранится в Z-буфере.
    if (zi < z_ptr[xi])
    {
        // Выводим пиксель текстуры, предполагая, что он
        // задан в формате 5.6.5.
        screen_ptr[xi] = ((ui>>(FIXP16_SHIFT+3))<<11) +
            ((vi>>(FIXP16_SHIFT+2))<<5) +
            ((wi>>(FIXP16_SHIFT+3)));

        // Обновляем Z-буфер.
        z_ptr[xi] = zi;
    } // if

    // Интерполируем величины u,v,w,z

```

```

    ui+=du;
    vi+=dv;
    wi+=dw;
    zi+=dz;
} // for xi

// Интерполируем величины u,v,w,z,х вдоль правой и левой
// сторон треугольника
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
wl+=dwdyl;
zl+=dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;
zr+=dzdyr;

// Сдвигаем указатель на экран
screen_ptr+=mem_pitch;

// Сдвигаем указатель на Z-буфер
z_ptr+=zpitch;
} // for y

```

Как в приводимом ранее фрагменте, код, **обрабатывающий Z-буфер**, выделен полужирным шрифтом. Интересно, что если пиксель не перезаписывается, то у нас получается некоторая экономия ресурсов. Напомним, что в старой функции растеризации пиксель визуализируется, а предназначенный для его вывода код выполняется независимо ни от чего. Однако в той версии, в которой поддерживается Z-буфер, обновление содержимого Z-буфера и вывод пикселя производится только в том случае, когда **текущее** значение z меньше хранящегося в Z-буфере. Поэтому в новой версии при занесении многоугольника в Z-буфер производится больше действий, зато, если Z-буфер обновлять не нужно, экономится время работы. Следует также учесть, что для поддержки интерполяции по Z-буферу достаточно было добавить две операции сложения во внешнем цикле.

```

zl+=dzdyl;
zr+=dzdyr;

```

и всего одну — во внутреннем, где осуществляется растеризация строки развертки.

```

zi+=dz;

```

Таким образом, в конечном счете Z-буфер не так уж плох! Знакомясь с демонстрационными программами, приятно удивляешься тому, что они работают почти так же быстро, что и те, в которых Z-буфер не использовался. Зато визуализация в них производится на уровне пикселей.

## Обновление функций, предназначенных для отображения текстуры

Напомним, что есть две функции, отображающие текстуру. Одна из них обрабатывает многоугольники с постоянным затенением, т.е. **вообще** не связана с источниками света. В этой функции на многоугольник просто накладывается текстура.

Другая функция обрабатывает многоугольники с текстурой и плоским затенением. В этой функции, имя которой заканчивается на “FS”, весь многоугольник сначала освещается белым светом, значение в формате RGB которого задается как (255,255,255). Далее этот цвет используется как базовый, и результирующий цвет образуется путем его модуляции цветами текстурной карты на уровне отдельных пикселей.

#### Функция для отображения текстуры с постоянным затенением

Поддержку Z-буфера нужно добавить в обе функции, производящие отображение текстуры. Как и в случае с обсуждавшейся перед этим функцией, это достаточно легко сделать. Надо только добавить еще одну интерполируемую величину, а также инструкцию сравнения величин z во внутреннем цикле — и все. Приведем новый прототип функции, предназначенной для отображения текстуры с постоянным затенением.

```
void Draw_Textured_TriangleZB16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR * _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,        // Количество байтов в
                        // строке: 320, 640 и т.д.
    UCHAR * _zbuffer,     // Указатель на Z-буфер
    int zpitch)           // Количество байтов в строке
                        // Z-буфера
```

Вызов функции производится точно так же, как и до этого, но теперь поверхности, передаваемой в функцию, нужно присвоить соответствующие значения координат z, а, кроме того, передать в функцию указатель на Z-буфер и длину его строки, выраженную в байтах. Как и для предыдущих функций, приводим фрагмент кода, имеющий отношение к Z-буферу, в котором полужирным шрифтом выделена та часть, где производится сравнение и обновление Z-буфера.

```
// Устанавливаем указатель на экран в начало строки
screen_ptr = dest_buffer + (ystart * mem_pitch);

// Устанавливаем указатель на Z-буфер в начало строки
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi <= yend; yi++)
{
    // Вычисляем расстояние между конечными точками
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные значения для интерполируемых
    // величин u, v
    ui = ul + FIXP16_ROUND_UP;
    vi = vl + FIXP16_ROUND_UP;
    zi = zl + FIXP16_ROUND_UP;

    // Вычисляем постоянные интерполяции для величин u, v и z
    if ((dx - (xend - xstart)) > 0)
    {
        du = (ur - ul) / dx;
        dv = (vr - vl) / dx;
        dz = (zr - zl) / dx;
    } // if
```

```

else
{
    du = (ur - ul);
    dv = (vr - vl);
    dz = (zr - zl);
} // else

////////////////////////////////////

// Проверяем, нужно ли выполнять отсечение по левому
// краю
if (xstart < min_clip_x)
{
    // Определяем перекрытие по x
    dx = min_clip_x - xstart;

    // Даем приращение интерполируемым величинам
    ui += dx * du;
    vi += dx * dv;
    zi += dx * dz;

    // Обновляем переменные
    xstart = min_clip_x;
} // if

// Проверяем, нужно ли выполнять отсечение по правому
// краю
if (xend > max_clip_x)
    xend = max_clip_x;

////////////////////////////////////

// Выводим строку развертки
for (xi = xstart; xi <= xend; xi++)
{
    // Проверяем, меньше ли текущая координата z того
    // значения, которое хранится в Z-буфере
    if (zi < z_ptr[xi])
    {
        // Выводим пиксель текстуры.
        screen_ptr[xi] =
            textmap[(ui >> FIXP16_SHIFT) +
                ((vi >> FIXP16_SHIFT) << texture_shift2)];

        // Обновляем Z-буфер.
        z_ptr[xi] = zi;
    } // if

    // Интерполируем величины u,v,z
    ui += du;
    vi += dv;

```

```

        zi+=dz;
    } // for xi

    // Интерполируем величины u,v,z,x вдоль правой и левой
    // сторон треугольника
    xl+=dxdyl;
    ul+=dudyl;
    vl+=dvdy;
    zl+=dzdyl;

    xr+=dxdyr;
    ur+=dudyr;
    vr+=dvdyr;
    zr+=dzdyr;

    // Сдвигаем указатель на экран
    screen_ptr+=mem_pitch;
    // Сдвигаем указатель на Z-буфер
    z_ptr+=zpitch;
} // for y

```

Как и раньше, все изменения сводятся к добавлению одной-двух интерполируемых величин во внутреннем и внешнем циклах растеризации с целью выполнить интерполяцию координат z.

### Функция для отображения текстуры с плоским затенением

А теперь рассмотрим функцию, производящую наложение текстуры на многоугольники с плоским затенением. Что касается объема кода, самой сложной является функция с поддержкой Z-буфера, реализующая затенение по **Гуро**, однако технически наиболее сложна функция, которую мы обсудим в данном разделе. Фактически нам понадобится создать полностью обновленную функцию, в которую можно передавать указатель на Z-буфер и его шаг. Приведем прототип этой функции.

```

void Draw_Textured_TriangleFSZB16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Количество байтов в
                        // строке: 320, 640 и т.д.
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch)         // Количество байтов в строке
                        // Z-буфера

```

Все работает так же, как и раньше; просто нужно убедиться, что функция вызывается с координатами z, правильно заданными для каждого многоугольника, и, конечно же, что в нее передается указатель на Z-буфер и длина его строки, выраженная в байтах. Наконец, приведем код цикла растеризации, в котором **осуществляется** поддержка Z-буфера (соответствующий фрагмент выделен полужирным шрифтом).

```

// Устанавливаем указатель на экран в начало строки
screen_ptr = dest_buffer + (ystart * mem_pitch);

// Устанавливаем указатель на Z-буфер в начало строки
z_ptr = zbuffer + (ystart * zpitch);

```

```

for (yi = ystart; yi <= yend; yi++)
{
    // Вычисляем расстояние между конечными точками
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные значения для интерполируемых
    // величин u,v
    ui = ul + FIXP16_ROUND_UP;
    vi = vl + FIXP16_ROUND_UP;
    zi = zl + FIXP16_ROUND_UP;

    // Вычисляем постоянные интерполяции для величин u,v и z
    if ((dx = (xend - xstart)) > 0)
    {
        du = (ur - ul)/dx;
        dv = (vr - vl)/dx;
        dz = (zr - zl)/dx;
    } // if
    else
    {
        du = (ur - ul);
        dv = (vr - vl);
        dz = (zr - zl);
    } // else

    // Выводим строку развертки
    for (xi = xstart; xi <= xend; xi++)
    {
        // Проверяем, меньше ли текущая координата z
        // значения, хранящегося в Z-буфере
        if (zi < z_ptr[xi])
        {
            // Выводим пиксель текстуры
            // Сначала извлекаем его
            textel =
                textmap[(ui >> FIXP16_SHIFT) +
                    (vi >> FIXP16_SHIFT) << texture_shift2]);

            // Извлекаем rgb-компоненты
            r_textel = ((textel >> 11));
            g_textel = ((textel >> 5) & 0x3f);
            b_textel = (textel & 0x1f);

            // Модулируем цвет пикселя текстуры цветом
            // фонового освещения
            r_textel *= r_base;
            g_textel *= g_base;
            b_textel *= b_base;

            // Наконец, выводим пиксель. Заметим, что
            // математические преобразования выполняются в
            // такой форме, что результаты имеют вид: r*32,

```

```

// g*64, b*32. Таким образом, их нужно разделить
// на 32,64 и 32 соответственно. Однако,
// поскольку потом их нужно сдвигать, чтобы
// образовать слово в формате 5.6.5, можно
// воспользоваться тем, что обе операции
// компенсируют друг друга. Кроме того,
// понадобится выполнить логическое сложение, но
// это будет сделано позже, после определенной
// оптимизации...
screen_ptr[xi] = ((b_textel >> 5) +
                  (g_textel >> 6) << 5) +
                  (r_textel >> 5) << 11));

// Обновляем Z-буфер
z_ptr[xi] = zi;
} // if

// Интерполируем величины u,v,z
ui+=du;
vi+=dv;
zi+=dz;
} // for xi

// Интерполируем величины u,v,z,x вдоль правой и левой
// сторон треугольника
xl+=dxdyl;
ul+=dudyl;
vl+=dvdy;
zl+=dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
zr+=dzdyr;

// Сдвигаем указатель на экран
screen_ptr+=mem_pitch;

// Сдвигаем указатель на Z-буфер
z_ptr+=zpitch;

} // for y

Внутренний цикл довольно сложный, поэтому то, что не нужно выводить пиксель,
когда условие сравнения в Z-буфере не выполняется, — это весьма позитивный момент.
В этом случае удастся избежать выполнения довольно объемного фрагмента кода, в кото-
ром производятся запутанные вычисления.

// Выводим пиксель текстуры
// Сначала его извлекаем
textel = textmap[(ui >> FIXP16_SHIFT)+
                 ((vi >> FIXP16_SHIFT) << texture_shift2)];

```

```

// Извлекаем rgb-компоненты
r_textel = ((textel >> 11));
g_textel = ((textel >> 5) & 0x3f);
b_textel = (textel & 0x1f);

// Модулируем цвет пикселя текстуры цветом фонового
// освещения
r_textel *= r_base;
g_textel *= g_base;
b_textel *= b_base;
screen_ptr[xi] = ((b_textel >> 5) +
                  ((g_textel >> 6) << 5) +
                  ((r_textel >> 5) << 11));

// Обновляем Z-буфер
z_ptr[xi] = zi;

```

Интересно отметить, что по мере усложнения функций растеризации, они, казалось бы, должны замедлять свою работу, но на самом деле оказывается, что эти функции работают **еще** быстрее. Это происходит благодаря тому, что в реальном игровом мире потери часто оборачиваются преимуществами. В данном случае, например, отпадает необходимость выводить пиксели, которые в силу их расположения не заносятся в Z-буфер, поэтому мы без затрат получаем возможность выполнять визуализацию на уровне пикселей.

### Обновление функции, обрабатывающей список визуализации

Еще одна функция, которую нужно обновить, чтобы завершить поддержку Z-буфера, — это функция, выполняющая обработку списка визуализации. В нее передается список визуализации, а затем в теле этой функции вызываются соответствующие функции растеризации (исходя из того, к какому типу принадлежит тот или ной многоугольник). Приведем новую версию функции, которая выводит содержимое списка визуализации.

```

void Draw_RENDERLIST4DV2_SolidZB16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR      *video_buffer,
    int        lpitch,
    UCHAR      *zbuffer,
    int        zpitch)
{
    // 16-битовая версия. Эта функция обрабатывает список
    // визуализации. Другими словами, она выводит на экран
    // все имеющиеся в этом списке поверхности. На основе
    // модели освещения, заданной для многоугольника, в этой
    // функции вызывается надлежащая функция растеризации

    POLYF4DV2 face; // Временная поверхность, использующаяся
                    // для визуализации многоугольника

    // На данном этапе уже есть список многоугольников, и
    // пора вывести их на экран
    for (int poly=0; poly < rend_list->num_polys; poly++)
    {
        // Данный многоугольник визуализируется только в том

```

```

// случае, если он не отсекается, не
// отбраковывается, если он активен и видим. Однако
// заметим, что концепция обратной поверхности для
// игрового процессора, работающего в каркасном
// режиме, не имеет смысла
if (!(rend_list->poly_ptrs[poly]->state &
    POLY4DV2_STATE_ACTIVE) ||
    (rend_list->poly_ptrs[poly]->state &
    POLY4DV2_STATE_CLIPPED) ||
    (rend_list->poly_ptrs[poly]->state &
    POLY4DV2_STATE_BACKFACE) )
    continue; // Переходим к следующему
               // многоугольнику

// Сначала необходимо проверить наличие текстуры,
// так как многоугольники с текстурой могут иметь
// либо постоянное, либо плоское затенение. В
// зависимости от этого вызывается та или иная
// функция растеризации
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
}

// Присваиваем вершинам соответствующие параметры
face.tvlist[0].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
face.tvlist[0].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].z;
face.tvlist[0].u0 =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].u0;
face.tvlist[0].v0 =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].v0;

face.tvlist[1].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].z;
face.tvlist[1].u0 =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].u0;
face.tvlist[1].v0 =
    (int)rend_list->poly_ptrs[poly]->tvlist[1].v0;

face.tvlist[2].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].z;
face.tvlist[2].u0 =

```

```

    (int)rend_list->poly_ptrs[poly]->tvlist[2].u0;
face.tvlist[2].v0 =
    (int)rend_list->poly_ptrs[poly]->tvlist[2].v0;

// Присваиваем текстуру
face.texture =
    rend_list->poly_ptrs[poly]->texture;

// Это текстура с постоянным затенением?
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
{
    // Выводим треугольник с текстурой
    // и постоянным затенением
    Draw_Textured_TriangleZB16(&face,
        video_buffer, lpitch, zbuffer,
        zpitch);
} // if
else
{
    // Выводим треугольник с плоским затенением
    face.lit_color[0] =
        rend_list->poly_ptrs[poly]->lit_color[0];
    Draw_Textured_TriangleFSZB16(&face,
        video_buffer, lpitch, zbuffer, zpitch);
} // else
} // if
else if ((rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
    (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
{
    // Выводим треугольник с постоянным затенением
    face.lit_color[0] =
        rend_list->poly_ptrs[poly]->lit_color[0];

    // Задаем параметры вершин
    face.tvlist[0].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].z;

    face.tvlist[1].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].z;

    face.tvlist[2].x =

```

```

        (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].z;

// Выводим треугольник с помощью основной
// функции, выполняющей растеризацию с плоским
// затенением
Draw_Triangle_2DZB_16(&face, video_buffer,
        lpitch, zbuffer, zpitch);

} // if
elseif (rend_list->poly_ptrs[poly]->attr &
        POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
{
    // {Позже воспользуемся преимуществами этих
    // структур данных..}
    // Задаем параметры вершин
    face.tvlist[0].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.lit_color[0] =
        rend_list->poly_ptrs[poly]->lit_color[0];

    face.tvlist[1].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.lit_color[1] =
        rend_list->poly_ptrs[poly]->lit_color[1];

    face.tvlist[2].x =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z =
        (int)rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.lit_color[2] =
        rend_list->poly_ptrs[poly]->lit_color[2];

    // Выводим треугольник с затенением по Гуро
    Draw_Gouraud_TriangleZB16(&face, video_buffer,
        lpitch, zbuffer, zpitch);
} // if gouraud
} // for poly
} // Draw_RENDERLIST4DV2_SolidZB16

```

Остановимся на минуту и задумаемся над тем, насколько простой и понятной является структура этой функции. Сначала в ней определяется, к какому виду относится текущий многоугольник, а затем вызывается соответствующая функция растеризации. Конечно же, если использовать все возможности языка C++, можно было бы применить перегрузку функций или воспользоваться виртуальными функциями и обойтись без структуры с несколькими ветвями. Однако преимущества C++ проявляются в тех случаях, когда конструкция программы фиксирована, а мы все время что-то изменяем, поэтому упомянутые возможности не очень-то нам помогут. Мы просто пишем код, экспериментируем с различными алгоритмами и смотрим, что происходит. Позже на основе полученных результатов можно будет разработать более понятный и эффективный код. Как бы то ни было, функция `Draw_RENDERLIST4DV2_SolidZB16()` работает точно так же, как и ее предыдущая версия `Draw_RENDERLIST4DV2_Solid16()`, только теперь она имеет дополнительные параметры: указатель на Z-буфер и длину его строки, выраженную в байтах.

#### НА ЗАМЕТКУ

Возможно, вас удивляет, что вместо самого объекта Z-буфера в каждую функцию передается указатель на него. Просто таким способом я хотел сохранить возможность обновления объекта Z-буфера `ZBUFFERV1`, не переделывая при этом функцию растеризации.

## Возможные оптимизации Z-буфера

Вычисления величин, хранящихся в Z-буфере, настолько просты, что нет смысла прибегать к ассемблеру, инструкциям SIMD и другим подобным приемам. Таким образом, возможность оптимизации нужно поискать на более высоком уровне.

### Сокращение объема используемой памяти

Первая возможность ускорить обработку Z-буфера — уменьшить его размеры, т.е. вместо 32-битового Z-буфера использовать 16-битовый. Несмотря на кажущуюся привлекательность этой идеи, доступ к 16-битовому Z-буферу может оказаться более медленным, чем к 32-битовому. Причина в том, что процессоры Pentium являются 32-битовыми устройствами, поэтому предпочтительными для них являются операции с 32-битовыми значениями. Кроме возрастания когерентности кэша при меньшем объеме памяти, приходящейся на строку, из 16-битового доступа можно извлечь выгоду при помощи особенностей выравнивания и других тонких приемов. Кроме того, 16 битов — это объем памяти, близкий к необходимому минимуму для представления любого десятичного значения, которым выражается разрешение по z, но это может не быть верным в конкретных ситуациях. Что касается лично меня, то я пришел к выводу, что, поскольку обработка Z-буфера производится во внутреннем цикле функции растеризации, преимущество применения 16-битового Z-буфера является незначительным.

### Менее частая очистка Z-буфера

Заполнение области памяти, в ходе которого производится очистка Z-буфера, требует определенного времени. По сути, это время, необходимое для записи в память 32-битового значения, умноженное на размер Z-буфера. В качестве примера предположим, что эта операция выполняется на компьютере с процессором Pentium III+, на котором операция записи в память из-за кэширования и других эффектов занимает в среднем четыре такта. Предположим также, что процессор работает с частотой 500 МГц. Таким

образом, один такт имеет длительность  $2 \cdot 10^{-9} \text{ с} = 2 \text{ нс}$ . Если буфер имеет размеры  $800 \times 600$ , то получаем полное время записи, равное  $4 \cdot 800 \cdot 600 \cdot 2 \cdot 10^{-9} \text{ с} = 3.84 \text{ мкс}$ .

Может показаться, что это не так уж много, но на самом деле это не так. При частоте обновления, равной 60 fps, время экспонирования одного кадра равно 16мс, так что получается, что примерно четверть всего времени тратится только на очистку Z-буфера, а такие потери уже не назовешь незначительными! Теперь уже не вызывает сомнений, что ситуация нуждается в улучшении. Для этого у нас есть несколько возможностей. Во-первых, можно попытаться грубо отследить прямоугольные области Z-буфера, которые не были изменены в кадре, и не очищать их. Этот прием выглядит неплохо, но на практике он не работает. Единственная альтернатива — не очищать Z-буфер вообще. Однако это можно выполнить, только если реализуется 1/z -буферизация. Поскольку в процессе 1/z -буферизации величина 1/z не превышает 1.0 при  $z > 1$ , можно смещать каждый кадр, добавляя сдвиг по оси z к каждому значению, хранящемуся в Z-буфере. Таким образом, каждый последующий кадр располагается немного дальше предыдущего, но при этом гарантируется, что в буфере кадры не будут мешать друг другу. Кроме того, следует изменить условие сравнения по z, поскольку значения z в каждом последующем кадре возрастают, но это реализуется простым изменением сравниваемых величин. Схема этого процесса приведена на рис. 11.10.

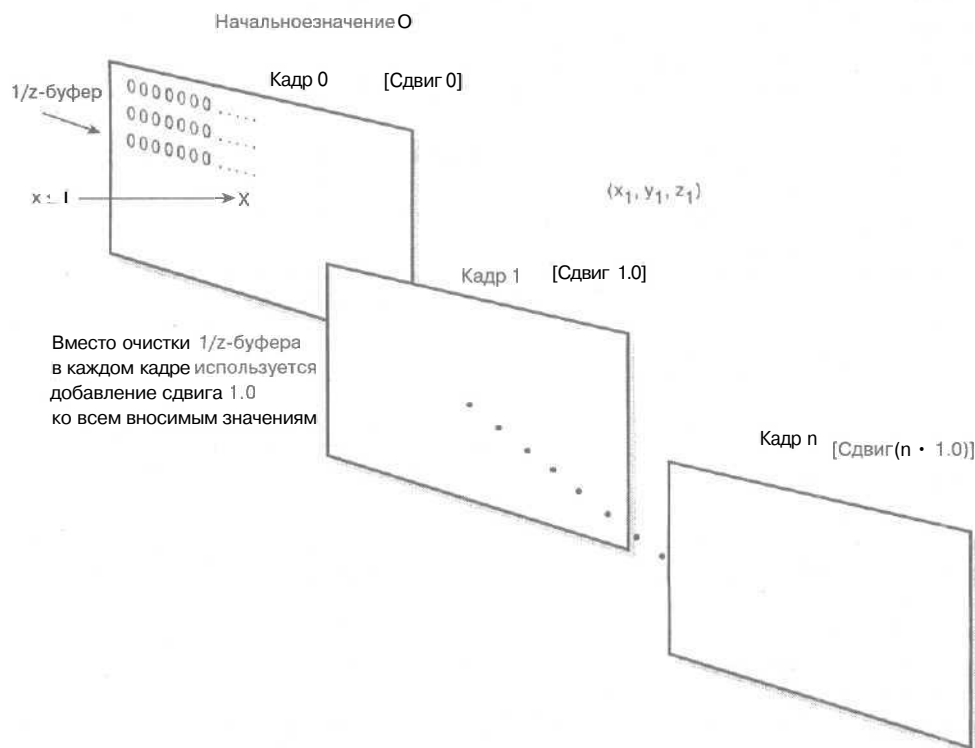


Рис. 11.10. Способ избежать очистки при 1/z -буферизации

В качестве примера рассмотрены два кадра, в которых используется этот метод. Координаты  $(x, y)$  у пикселей этих кадров остаются одними и теми же, и нужно следить только за координатой  $z$ . Наборы пикселей каждого кадра выводятся поверх пикселей преды-

душего кадра с другими значениями координат  $z$ , чтобы обеспечить в результате правильность содержимого  $1/z$ -буфера. Таким образом, если в буфере хранятся значения  $1/z$ , очередной кадр выводится правильно. Давайте проведем эксперимент. Пусть есть кадр с представленными ниже параметрами.

Кадр 0: начальное значение  $Z$ -буфера равно 0.0

Номер пикселя	$z$	$1/z$	Сдвиг	Конечное значение $1/z$ (с учетом сдвига)
0	10	0.100	0	0.100
1	12	0.083	0	0.083
2	5	0.200	0	0.200

#### Анализ кадра 0

$Z$ -буфер = 0.0

Конечное значение величины  $1/z$  для пикселя 0 равно 0.1, а это больше, чем 0.0. Поэтому этот пиксель выводится на экран, а величина в  $1/z$ -буфере обновляется значением 0.1.

$Z$ -буфер = 0.1

Конечное значение величины  $1/z$  для пикселя 1 равно 0.083, что **меньше**, чем 0.1. Поэтому пиксель НЕ записывается, а содержимое  $1/z$ -буфера остается неизменным.

$Z$ -буфер = 0.1

Конечное значение величины  $1/z$  для пикселя 2 равно 0.2, что превышает 0.1. Соответственно, данный пиксель выводится на экран, а величина в  $1/z$ -буфере обновляется значением 0.2.

Суть представленного выше анализа в том, что пиксель, ближе других расположенный к точке наблюдения (это пиксель 2 с координатой  $z$ , равной 5.0), следует выводить на экран и записывать в  $Z$ -буфер последним. На самом деле все так и происходит.

Кадр 1: начальное значение в  $Z$ -буфере равно 0.2

Номер пикселя	$z$	$1/z$	Сдвиг	Конечное значение $1/z$ (с учетом сдвига)
0	5	0.200	1.0	1.200
1	2	0.500	1.0	1.500
2	3	0.333	1.0	1.333

#### Анализ кадра 1

$Z$ -буфер = 0.0

Конечное значение величины  $1/z$  для пикселя 0 равно 1.2, что больше, чем 0.2. Поэтому данный пиксель выводится на экран, а величина в  $1/z$ -буфере получает значение 1.2.

$Z$ -буфер = 1.2

Конечное значение величины  $1/z$  для пикселя 1 равно 1.5, что больше, чем 1.2. Поэтому этот пиксель выводится на экран, а величина в  $1/z$ -буфере обновляется значением 1.5.

$Z$ -буфер = 1.5

Конечное значение величины  $1/z$  для пикселя 2 равно 1.333, что меньше, чем 1.5. Поэтому пиксель НЕ записывается, а содержимое  $1/z$ -буфера остается неизменным.

$Z$ -буфер = 1.5

Итак, в результате анализа выясняется, что, как и следует, пиксель, ближе других расположенный к точке наблюдения (это пиксель 1, которому соответствует координата  $z$ , равная 2.0), следует выводить на экран и записывать в  $Z$ -буфер последним.

Итак,  $1/z$ -буфер с отложенной очисткой работает, и работает правильно. Единственная проблема заключается в том, что в некоторый момент в  $1/z$ -буфере может возникнуть переполнение. Это происходит после смены  $p$  кадров, где  $p$  — максимально допустимое целое число, которое может быть представлено целой частью  $1/z$ -буфера.

## Смешанная Z-буферизация

Еще одна возможность оптимизировать Z-буфер — использовать смешанную Z-буферизацию. Другими словами, в Z-буфер заносится только определенная часть объектов, участвующих в игровой сцене, а для остальных — производится Z-сортировка. Например, можно было бы отбирать многоугольники, удаленные от точки наблюдения больше, чем на 30% от величины видимой области, и выполнять для них Z-сортировку, поскольку в большинстве случаев они все равно будут выглядеть удовлетворительно. Однако многоугольники, попадающие в 30-процентную зону области видимости, заносятся в Z-буфер. Этот прием работает замечательно. Сначала выполняется Z-сортировка удаленных многоугольников и их визуализация в порядке уменьшения расстояния до точки наблюдения по алгоритму художника, а потом с помощью Z-буфера обрабатываются ближние многоугольники, которые должны выглядеть безукоризненно. Описанный метод проиллюстрирован на рис. 11.11.

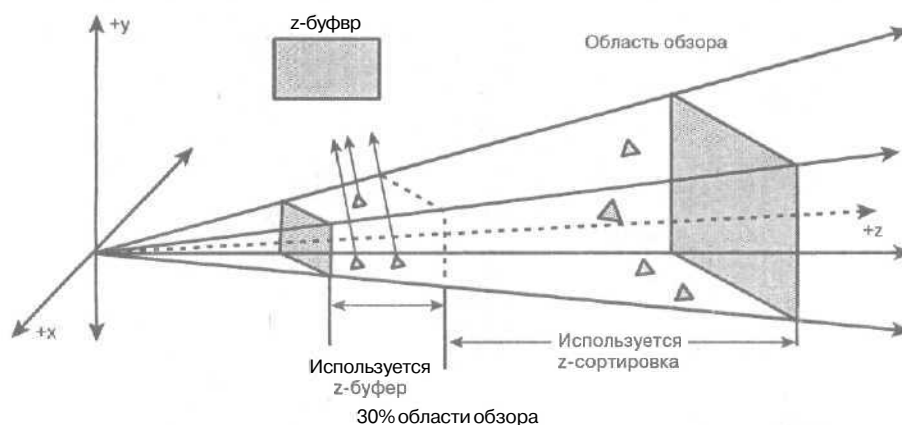


Рис. П. 11. Смесь Z-буфера и Z-сортировки

## Проблемы, связанные с Z-буфером

Напоследок я хочу обсудить некоторые проблемы, связанные с Z-буфером. Очевидная проблема заключается в том, что он работает неправильно. Мы интерполируем величину, которая в экранном пространстве изменяется по нелинейному закону, и поэтому сразу же рискуем получить различные аномалии, однако и это не главное. Основная связанная с Z-буфером проблема заключается в том, что для области видимости большого размера мы рискуем потерять точность из-за аксонометрического преобразования и нелинейности изменения интерполируемой величины  $z$ . Поясним сказанное на примере. Представьте, что игровое пространство ограничивается дальней плоскостью отсечения, расположенной на расстоянии 1 000 000 единиц от точки наблюдения (для космических игр это вполне разумная величина). Для точного представления 1 000 000 единиц понадобится около 20 битов. Даже если Z-буфер 32-битовый, для дробной части координаты  $z$

остается всего 12 битов. Таким образом, вся точность величин Z-буфера расходуется на представление **объектов**, расположенных вдоль этого огромного пространства, в то время как она нужна нам в **первую** очередь для описания близких фрагментов игрового пространства. Это еще одна причина, по которой лучше использовать **1/z-буфер**.

## Демонстрационные программы, использующие Z-буферизацию

Сначала поговорим о новых программных модулях. Весь код, разработанный в данной главе, содержится в новых библиотечных файлах:

- T3DLIB9.CPP — исходный C/C++ файл;
- T3DLIB9.H — заголовочный файл для T3DLIB9.CPP.

Конечно, для компиляции программ понадобятся библиотеки T3DLIB1-8.CPP|H, а также новая библиотека T3DLIB9.CPP|H.

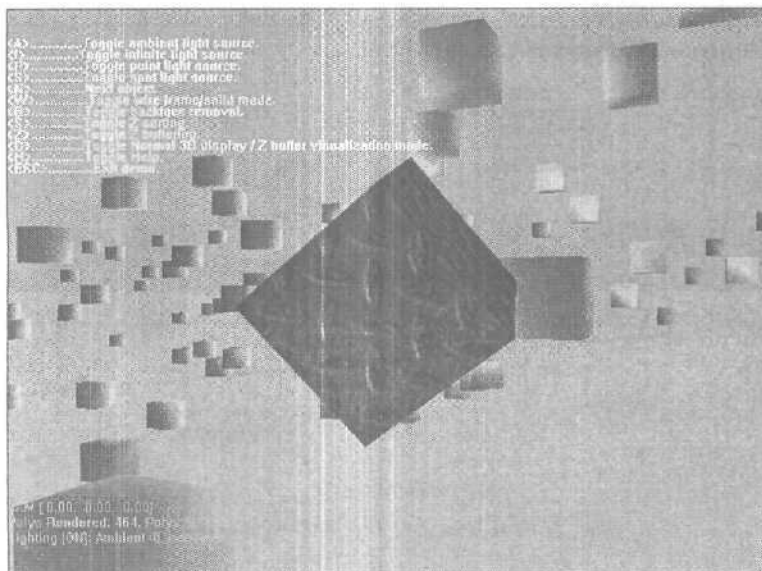


Рис. 11.12. Копия экрана первой демонстрационной программы, в которой используется Z-буфер

## Первая демонстрационная программа: визуализация Z-буфера

Как обычно, для закрепления материала главы хотелось бы привести пример игровой демонстрационной программы. Начну я все же с технического примера. На рис. 11.12 приведена копия экрана демонстрационной программы DEMO111\_1.CPP|EXE. В этой программе, как и в других ей подобных, с помощью клавиш со стрелками можно медленно перемещаться в пространстве. Замечательной особенностью программы является то, что с **помощью** клавиши <Z> можно выбирать алгоритм сортировки многоугольников вдоль

оси *z* — *Z*-сортировку или *Z*-буферизацию. Попробуйте поочередно выбирать то один способ сортировки, то другой, а затем, перемещаясь в игровом пространстве, понаблюдайте за результатами применения *Z*-сортировки и *Z*-буфера. Нетрудно заметить, что версия, в которой используется *Z*-буфер, работает просто превосходно. Было бы, однако, еще интереснее, если бы можно было визуализировать *Z*-буфер.

На самом деле это очень трудно. Для этого я копировал содержимое *Z*-буфера в буфер кадров с помощью простого трансляционного фильтра, который воспринимает 32-битовые значения глубины по оси *z* и преобразует их в значения **цветов**. Поскольку цвета в формате RGB кодируются 16-битовыми значениями, в старшие биты которых заносится интенсивность красного цвета, а в младшие — значение синего цвета, содержимое *Z*-буфера можно представить как цветовую карту **глубины**, в которой красный цвет соответствует большим расстояниям, а синий — объектам, расположенным недалеко от точки наблюдения (рис. 11.13).

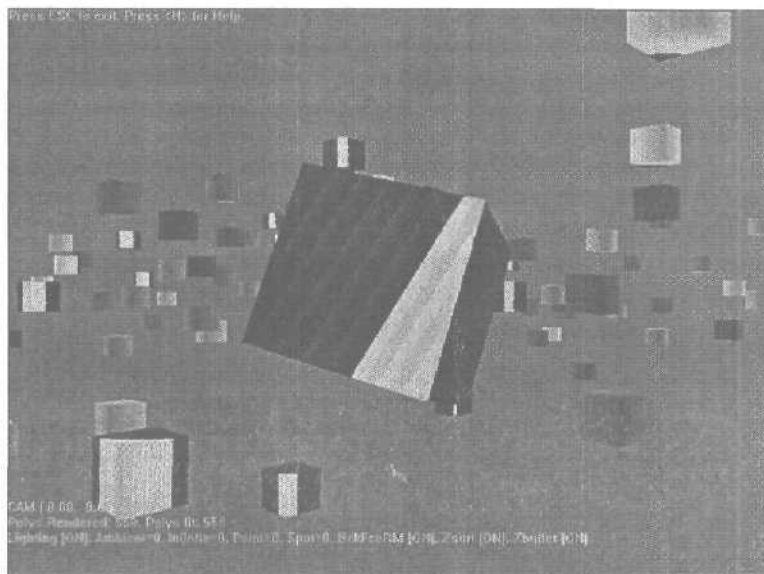


Рис. 11.13. Работа демонстрационной программы в режиме визуализации *Z*-буфера

Перечислим самые важные управляющие клавиши демонстрационной программы:

- клавиши со стрелками — перемещение;
- <Z> — переключение режима *Z*-буферизации;
- <S> — переключение режима *Z*-сортировки;
- <N> — выбор объектов;
- <H> — включение и выключение справки;
- <D> — выбор между обычной визуализацией и визуализацией *Z*-буфера;
- <Esc> — выход из программы.

Чтобы скомпилировать эту демонстрационную программу, понадобится ее исходный файл `DEMO111_1.CPP`, а также все библиотечные модули `T3DLIB1-9.CPP`. Кроме того, как обычно, нужно будет подключить библиотеки `DirectX` и `WINMM.LIB`.

В режиме визуализации Z-буфера можно заметить эффект цикличности цветов. Этот эффект связан с тем, что координата  $z$  принимает значения в интервале от величины, соответствующей точке наблюдения, до бесконечности. Однако поскольку цвета кодированы в формате RGB 5.6.5, при отображении координаты  $z$  (которая *изменяется* от 0 до 65536) на пространство цветов в младших битах, представляющих синий цвет, интенсивность цвета будет изменяться с периодом  $2^5=32$ . Таким образом, одна и та же интенсивность синего цвета будет сформирована 2048 раз. Тем не менее, на экране четко прослеживается глобальная закономерность: смещение *оттенка* цвета от синего к красному соответствует увеличению расстояния от точки наблюдения.

## Вторая демонстрационная программа: водный мотоцикл

Вторую демонстрационную программу мне захотелось создать на основе демонстрационной программы с ландшафтом из предыдущей главы. Я несколько улучшил физическую модель и получил симулятор водного мотоцикла. Вода — замечательный объект для демонстрации работы Z-буфера, поскольку многоугольники, из которых состоит поверхность какого-либо судна, должны иметь возможность проникать сквозь многоугольники, составляющие водную поверхность, благодаря чему имитируется частичное погружение. Обратите внимание на рис. 11.14, на котором изображена копия экрана демонстрационной программы DEMO11\_2.CPP|EXE. Для того объема кода, который занимает программа, изображение выглядит вполне сносно.

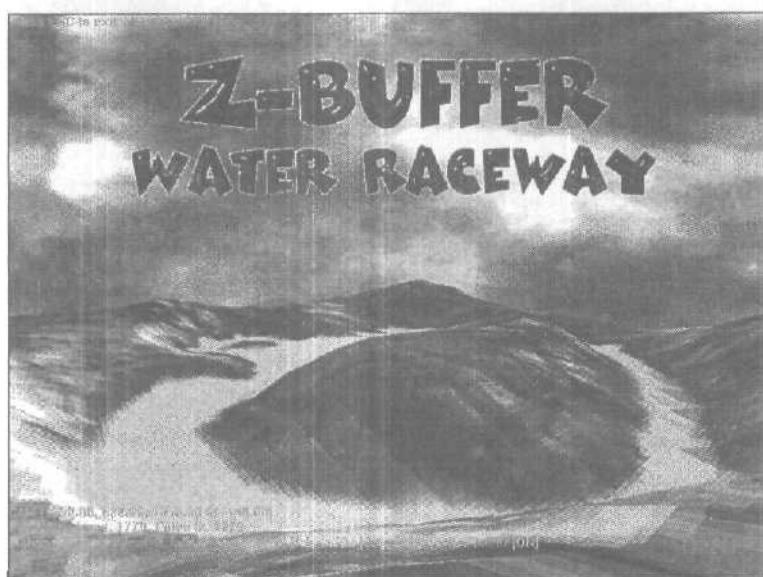


Рис. 11.14. Использование Z-буфера позволяет хорошо смоделировать воду

Приведем список управляющих клавиш программы:

- <Enter> — запуск двигателя;
- клавиша со стрелкой вправо — поворот направо;
- клавиша со стрелкой влево — поворот налево;
- клавиша со стрелкой вверх — газ.

- <Z> — переключение режима Z-буферизации;
- <S> — переключение режима Z-сортировки;
- <D> — выбор между обычной визуализацией и визуализацией Z-буфера;
- <H> — включение и выключение справки;
- <Esc> — выход из программы.

## Генерация ландшафта

- Terragen —<http://www.planetside.co.uk/terrigen/>
- Worldmachine —<http://students.washington.edu/sschmitt/world/>

The screenshot shows the NetLogo environment with a network diagram. The interface includes a menu bar (File, Edit, View, etc.), a toolbar, and a command area on the left. The main workspace displays a network of nodes and directed edges. Nodes are labeled: 'G3 power master', 'G3 correct signal', 'G3 correct trigger', 'G3 correct sensor', 'G3 correct', 'G3 condition', 'G3 trigger', 'G3 sensor', 'G3 correct trigger', and 'G3 R2 Output'. The nodes are interconnected with arrows indicating the flow of information or control.

С помощью этой профаммы была сгенерирована карта высот, а также текстура, отображаемая на ландшафт. На рис. 11.16 изображена карта высот, представленная в шкале уровней серого цвета, а на рис. 11.17 — замечательная карта текстуры, сгенерированная

программой **Worldmachine**. В игре использована карта высот ландшафта размером 40x40 и текстура размером 256x256, однако на рисунках проиллюстрированы версии с более высоким разрешением 512x512.

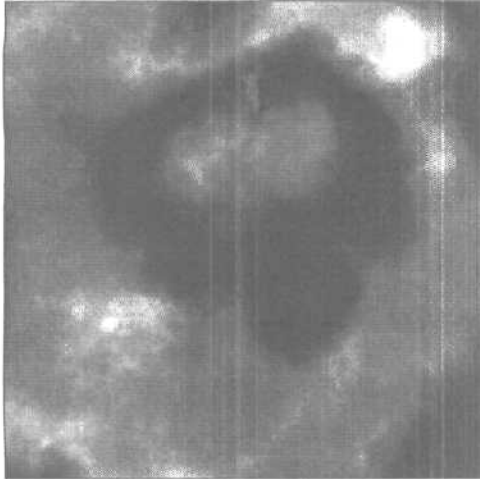


Рис. 11.16. Карта высот



Рис. 11.17. Карта текстуры водного канала

После того как ландшафт был готов, я вызвал нашу функцию, генерирующую каркас ландшафта.

```
// Генерируем ландшафт
Generate_Terrain_OBJECT4DV2(
    &obj_terrain, // Указатель на объект.
    TERRAIN_WIDTH, // Ширина в мировых координатах по оси x
    TERRAIN_HEIGHT, // Длина в мировых координатах по оси z
    TERRAIN_SCALE, // Масштабный множитель по вертикали
    "water_track_height_04.bmp", // Имя файла карты высот,
    // закодированной с помощью 256 цветов
    "water_track_color_03.bmp", // Имя файла с картой
    // текстуры
    RGB16Bit{255,255,255}, // Цвет ландшафта при
    // отсутствии текстуры
    &terrain_pos, // Начальное положение
    NULL, // Начальная ориентация
    POLY4DV2_ATTR_RGB16)
POLY4DV2_ATTR_SHADE_MODE_FLAT |
/*POLY4DV2_ATTR_SHADE_MODE_GOURAUD */
POLY4DV2_ATTR_SHADE_MODE_TEXTURE );
```

Как видите, ландшафт имеет плоское затенение.

### Анимация водной поверхности

Сделать поверхность воды подвижной очень просто. Идея заключается в том, чтобы модулировать каркас ландшафта волнообразно изменяющейся функцией, например синусом. Весь фокус в том, чтобы модулировать только ту часть ландшафта, которая имитирует поверхность воды. Когда начинается формирование очередной кадр, можно начать с исходного каркаса в локальных координатах, а затем скопировать его в преобразован-

ный каркас, который подлежит изменению. После этого запускается функция, генерирующая волны (вскоре мы с ней ознакомимся), которая обрабатывает весь каркас ландшафта. Однако модуляция применяется только к тем вершинам, которые находятся ниже виртуального уровня воды. В этом и состоит трюк. Описанный выше прием проиллюстрирован на рис. 11.18.

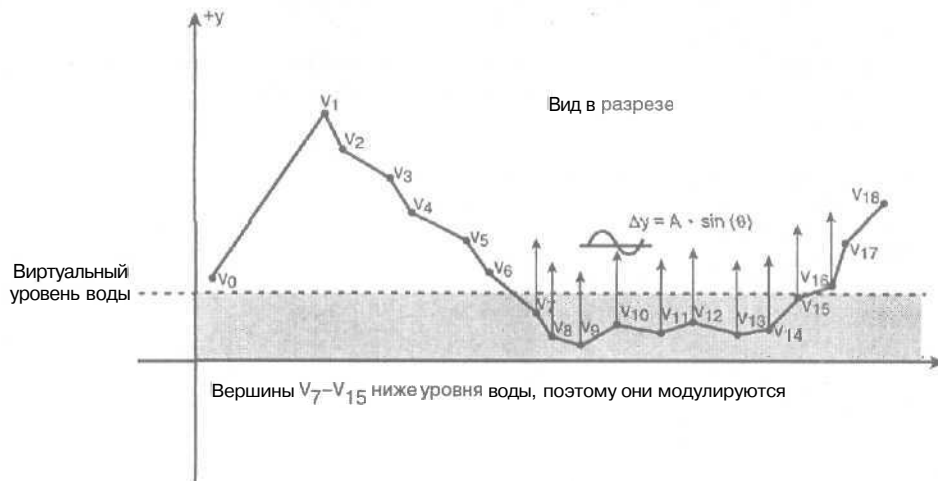


Рис. 11.18. Модулируется только та часть ландшафта, которая находится ниже уровня воды

Приведем код функции, генерирующей волны только в той части ландшафта, которая находится ниже уровня воды.

```
// Генератор волн

// Модуляция применяется для всех вершин ландшафта,
// для которых высота ниже уровня воды
for (int v = 0; v < obj_terrain.num_vertices; v++)
{
    // Волны генерируются только ниже уровня воды
    if (obj_terrain.vlist_trans[v].y < WATER_LEVEL)
        obj_terrain.vlist_trans[v].y +=
            WAVE_HEIGHT * sin(wave_count +
                (float)v / (2 * (float)obj_terrain.ivar2 + 0));
} // for v

// Увеличение волновой составляющей со временем
wave_count += WAVE_RATE;
```

Все это выглядит отвратительно, не правда ли? Возможно, вы подумали, что нам потребуется какая-то сложная физика, коэффициенты жесткости, волновые функции, — нет! Все, что нам нужно, — это синусообразная волна и счетчик. Нельзя сказать, что разработка реалистичной модели воды не важна, однако иногда для создания спецэффектов можно обойтись предельно простым кодом.

### Физическая модель водного мотоцикла

Физическая модель водного мотоцикла идентична модели вездехода, использовавшейся в предыдущей главе. Изменились только константы модели, в результате чего езда мотоцикла

стала более “пружинящей”, и теперь он способен глубже погружаться в воду, чтобы продемонстрировать преимущества Z-буфера. Как уже было сказано, физическая модель в основном осталась неизменной. Игрок жмет на газ, мотоцикл ускоряется, и по мере его движения отслеживается уровень того участка, на котором он находится. В соответствии с этим модель транспортного средства сдвигается вверх или погружается в воду. Замечательной особенностью данной игровой программы является то, что в результате волнообразного движения вершин каркаса водный мотоцикл тоже как будто качается на волнах! Приведем код, в котором реализована общая физическая модель, а также представлено размещение модели водного мотоцикла перед камерой (правда, пока это выглядит несколько сыровато).

```
// Имитация движения

// Отслеживаем неровности ландшафта. Просто определяем, над
// какой ячейкой находится мотоцикл, затем по индексу в
// списке вершин находим 4 вершины, образующие эту ячейку, и
// усредняем их высоты. На основе соотношения между текущим
// значением высоты и высотой ландшафта мотоцикл сдвигается
// по высоте

// В процессе генерирования ландшафта некоторые результаты
// сохраняются, чтобы можно было отслеживать неровности
// поверхности
//ivar1=columns;
//ivar2 - rows;
//fvar1 - col_vstep;
//fvar2 - row_vstep;

int cell_x - (cam.pos.x + TERRAIN_WIDTH/2) /
    obj_terrain.fvar1;
int cell_y - (cam.pos.z + TERRAIN_HEIGHT/2) /
    obj_terrain.fvar1;

static float terrain_height, delta;

// Проверяем, находится ли мотоцикл на поверхности
if ((cell_x >=0) && (cell_x < obj_terrain.ivar1) &&
    (cell_y >=0) && (cell_y < obj_terrain.ivar2))
{
    // Определяем индексы вершин текущего четырехугольника
    int v0 - cell_x + cell_y*obj_terrain.ivar2;
    int v1 - v0 + 1;
    int v2 - v1 + obj_terrain.ivar2;
    int v3 - v0 + obj_terrain.ivar2;

    // Теперь просто обращаемся по этому индексу к таблице
    terrain_height = 0.25 * (obj_terrain.vlist_trans[v0].y +
        obj_terrain.vlist_trans[v1].y +
        obj_terrain.vlist_trans[v2].y +
        obj_terrain.vlist_trans[v3].y);

    // Вычисляем разность высот
    delta - terrain_height - (cam.pos.y - gclearance);

    // Проверяем, не произошло ли проникновение
    if (delta > 0)
    {
```

```

// Тут же применяем силу к камере
// (имитируем выталкивающее действие воды)
vel_y+=(delta * (VELOCITY_SCALER));

// Проверяем, не произошло ли проникновение. Если
// мотоцикл находится ниже уровня поверхности, сразу
// сдвигаем его вверх
cam.pos.y+=(delta*CAM_HEIGHT_SCALER);

// Это больше похоже на трюк, чем на физическую
// модель — на основе скорости мотоцикла и градиента
// высот организуем небольшую качку камеры
cam.dir.x - (delta* PITCH_CHANGE_RATE);

} // if

} // if

// Замедление камеры
if (cam_speed > (CAM_DECEL) ) cam_speed-=CAM_DECEL;
else if (cam_speed < (-CAM_DECEL)) cam_speed+=-CAM_DECEL;
else cam_speed - 0;

// Поиск стабильной ориентации камеры
if (cam.dir.x > (neutral_pitch+PITCH_RETURN_RATE))
    cam.dir.x - (PITCH_RETURN_RATE);
else if (cam.dir.x < (neutral_pitch-PITCH_RETURN_RATE))
    cam.dir.x += (PITCH_RETURN_RATE);
else
    cam.dir.x - neutral_pitch;

// Применяем гравитацию
vel_y+=gravity;

// Проверяем, не опустился ли вездеход ниже уровня воды, и
// при необходимости выталкиваем его на поверхность
if (cam.pos.y < sea_level)
{
    vel_y - 0;
    cam.pos.y - sea_level;
} // if

// Сдвигаем камеру
cam.pos.x += cam_speed*Fast_Sin(cam.dir.y);
cam.pos.z += cam_speed*Fast_Cos(cam.dir.y);
cam.pos.y += vel_y;

// Помещаем точечный источник перед игроком
lights2[POINT_LIGHT_INDEX].pos.x = cam.pos.x +
    150*Fast_Sin(cam.dir.y);
lights2[POINT_LIGHT_INDEX].pos.y = cam.pos.y + 50;
lights2[POINT_LIGHT_INDEX].pos.z = cam.pos.z +
    150*Fast_Cos(cam.dir.y);

// Помещаем игровой объект в воду чуть впереди камеры
obj_player.world_pos.x - cam.pos.x +

```

```

120*Fast_Sin(cam.dir.y);
obj_player.world_pos.y = cam.pos.y - 75 +
7.5*sin(wave_count);
obj_player.world_pos.z = cam.pos.z +
120*Fast_Cos(cam.dir.y);

```

Если бы мы писали реальную игру, то разместили бы и зафиксировали камеру поближе к игроку, а не использовали бы ее как основной объект с **последующей** фиксацией игрового объекта перед ней. Кроме того, в нашей физической модели не учитывается трение. Его легко добавить, но нужно же что-нибудь оставить и для других демонстрационных программ!

## Звуковые эффекты

Я горжусь звуком в этой игре. Я воспользовался несколькими образцами звуков, включая шум водного мотоцикла, плеск воды, шум океана, всплески и свой собственный голос (в качестве **ведущего**), и скомпоновал их так, чтобы вы могли получить опыт в создании игр. Звуковые эффекты обладают тремя различными аспектами. Дадим их краткую характеристику, поскольку звук — это важная составляющая игры, и он может внести в нее оживление или сделать ее скучной.

### Голос ведущего

Данная игра состоит из трех основных частей: спуск на воду, вывод инструкций по управлению и сами игровые действия. В начале игры звучит ее краткое описание. Затем игровой объект опускается на воду, после чего даются новые инструкции.

Чтобы создать звуковое сопровождение, я записал свой голос с частотой оцифровки 11 кГц в 8-битовом монорежиме. Для записи использовалась программа Sound Forge. Я добавил немного шумов, чтобы повысить степень реалистичности, и наконец добавил небольшое эхо. Результаты оказались **замечательными**, и даже мой друг не мог поверить, что это мой голос!

### Звуки воды

Для имитации звуков воды я воспользовался образцами из библиотеки звуков Sound Ideas General 6000. В ее состав входит 40 компакт-дисков, за что мне, правда, пришлось заплатить **около 2500 долл.**, однако оно того стоит. Я взял образцы тихого и громкого плеска волн на озере, а затем воспроизвел их с разными амплитудами и слегка модифицировал их частоты в зависимости от скорости игрока. Удивительно, но в 8-битовом монорежиме с частотой оцифровки 11 кГц **звуки** остались практически неизменными. **Звуки** воды зациклены, чтобы они могли воспроизводиться бесконечно долго.

### Звуки водного мотоцикла

Это самая сложная часть модели. Надо сказать, это всегда проблематично — добиться реалистичности при создании звуковых эффектов. В реальной жизни многие предметы не издают звуков — это просто нам кажется. В фильмах, например, все звуки до смешного преувеличены. Вот почему недалёковидные звукорежиссеры иллюстрируют звуком каждое событие в фильме, чтобы звучание не уступало изображению, даже если слышать нечего!

В любом случае водный мотоцикл может находиться в одном из следующих состояний:

1. двигатель выключен;
2. запуск двигателя (однократное звучание фрагмента);
3. холостой ход двигателя (звуковой фрагмент **зациклен**);
4. ускорение (однократное звучание фрагмента);
5. работа двигателя на повышенных оборотах (звуковой фрагмент зациклен).

Мне нужен был хороший звук запускаемого двигателя. В результате я остановился на звучании внешнего бортового двигателя 25-НР, смешанного со звуком запускаемого двигателя самого водного мотоцикла (иначе он был слишком приглушенным). Для имитации работы двигателя на холостых оборотах, а также ускоряющегося и работающего на повышенных оборотах двигателя были использованы звуковые фрагменты, записанные с помощью реального водного мотоцикла. Трудность состояла в том, чтобы правильно скомпоновать все эти звуки. Для этого нужно отслеживать состояние водного мотоцикла, его скорость и значения нескольких других переменных, сигнализирующих об изменении положения. Например, когда игрок нажимает клавишу <Enter> и запускает двигатель, включаются все звуки, кроме звучания ускоряющегося мотоцикла. При этом в основном слышен звук запускаемого двигателя, потому что он самый громкий, однако на заднем плане все же можно расслышать звучание двигателя на холостом ходу. Интенсивность звука, соответствующего состоянию 5, в этот момент равна нулю, потому что мотоцикл не двигается.

Когда игрок нажимает на газ, подключается звук ускоряющегося мотоцикла. Это всего один звуковой фрагмент, на протяжении которого слышно, как это транспортное средство трогается с места. А тем временем игровая модель водного мотоцикла ускоряется, и ее скорость используется для модулирования частоты и амплитуды звука работающего двигателя. Чем быстрее мы передвигаемся, тем больше высота и интенсивность звучания. Тем не менее, на заднем плане все же слышен звук двигателя, работающего на холостом ходу. Он как бы ожидает, когда игрок сбросит газ.

Конечно же, эти звуки можно смоделировать различными способами; добиться того, чтобы в каждый момент включался нужный звук, можно с помощью оптимизации, а не путем изменения громкости, однако все хорошо работает и без этих сложностей.

#### ВНИМАНИЕ

В функциях растеризации все еще может возникнуть ошибка деления на ноль. В определенных ситуациях в программе генерируются многоугольники, при обработке которых возникает сбой! Это изредка происходит при пересечении наивысшего пика, покрытого зеленой травой, в чем при желании вы можете убедиться и сами.

Чтобы скомпилировать эту демонстрационную программу, понадобится исходный файл с основным модулем DEMO11\_2.CPP, а также библиотечные модули T3DLIB1-9.CPP\H. Кроме того, как обычно, нужно подключить библиотеки DirectX и WINMM.LIB.

## Резюме

Чтобы освоить описанные методы, нужно просто затратить немного времени, разобрав примеры на бумаге, понять их — и все заработает. Главное — помните, что все начинается с вывода точки.

Итак, теперь у нас есть довольно приличный трехмерный игровой процессор! Он поддерживает 16-битовый режим, освещение, Z-буферизацию и загрузку моделей. Кроме того, он довольно быстро работает, особенно если учесть полное отсутствие оптимизации. Пока что я вполне им доволен. В дальнейшем игровой процессор будет пополняться дополнительными функциональными возможностями: улучшенным освещением, поддержкой иерархии объектов и т.п. На данный момент нам уже удалось освоить 90% всего того, что необходимо знать о трехмерной графике. Вам есть чем гордиться!





# ЧАСТЬ IV

## Секреты трехмерной визуализации

**В этой части...**

**Глава 12**

Методы сложного текстурирования 1015

**Глава 13**

Алгоритмы разбиения пространства  
и определения видимости 1143

**Глава 14**

Освещение и тени 1263



# ГЛАВА 12

## Методы сложного текстурирования

### В этой главе...

• Текстурирование — вторая волна	1016
• Построение нового базового растеризатора	1026
• Прозрачность и альфа-смешивание	1043
• Текстурирование с корректной перспективой и 1/z-буферизация	1068
• Билинейная фильтрация текстуры	1103
• Множественное отображение и трилинейная фильтрация текстур	1108
• Многопроходная визуализация и Текстурирование	I 33
• Все в одном вызове	1134

В этой главе мы обсудим различные методы растеризации. Хотя у нас уже есть довольно сложный программный процессор растеризации, в нем все еще нет поддержки альфа-смешивания, текстур с корректно обрабатываемой аксонометрией, текстурной фильтрации, текстур с затенением по Гуро и т.д. В данной главе мы и займемся этими вещами. Наконец, мы еще раз перепишем все растеризаторы. Вот основные темы этой главы:

- прозрачность и альфа-смешивание;
- Текстурирование с затенением по Гуро;
- Текстурирование с корректной перспективой;
- 1/2 -буферизация;

- билинейная фильтрация текстур;
- множественное отображение и трилинейная фильтрация текстур;
- новые загрузчики объектов;
- новая функция инициализации DirectDraw;
- многопроходная визуализация.

## Текстурирование — вторая волна

Мне пришлось написать более 50000 строк кода только для реализации растеризации — дело в том, что при любом изменении возможностей игрового процессора необходимо создавать новый растеризатор. Именно поэтому я отказался от идеи растеризатора **общего** назначения и его оптимизации.

После написания всех этих растеризаторов и демонстрационных программ для них я понял, что данная глава — не лучшее место для обсуждения вопросов освещенности и затенения: это совершенно отдельная тема. Кроме того, я хочу показать на конкретных примерах отображение освещенности, отображение среды, а также затенение. Чтобы сделать это **надлежащим** образом, мне необходимы средства для создания уровней.

В данной главе мы собираемся рассмотреть ряд новых методов, таких как инверсная Z-буферизация, альфа-смешивание, **текстурирование** с корректной перспективой, множественное отображение, билинейное и трилинейное текстурирование и другие. И конечно, нам нужно будет еще раз переписать каждый растеризатор, и не раз. В этой главе будет более 40 новых растеризаторов! В конечном итоге, я думаю, вы будете удивлены скоростью работы приложения, учитывая, что мы достигли этого без помощи ассемблера или особой оптимизации (мы займемся этим в конце книги).

Как обычно, у нас появится новый библиотечный модуль и заголовочный файл, с **помощью** которых нужно будет компилировать все программы этой главы:

**T3DLIB10.CPP** — файл исходным текстом на C/C++;

**T3DLIB10.H** — заголовочный файл.

Для компиляции любой программы из этой главы вам нужно включить в свой проект эти файлы вместе с ранее разработанными библиотечными модулями **T3DLIB1-9.CPP|H**. Вам также понадобятся **.LIB-файлы** DirectX, и конечно, нужно будет правильно настроить компилятор (указав в качестве целевых приложений Win32 .EXE). Я всегда указываю в начале кода, какой тип программы нужно компилировать, так что вы всегда можете узнать об этом из самого файла с исходным текстом программы.

В любом случае мы собираемся подробно рассмотреть каждую тему, проанализировать код, создать демонстрационную программу и двигаться дальше. Я не реализовал некоторые из методов (или реализовал и нашел их слишком медленными), например, такие как трилинейная фильтрация. Тем не менее, мы рассмотрим их все без исключения.

Кроме того, я обновил некоторые другие функции для поддержки новых возможностей — такие как функцию генерации ландшафтов, загрузчик **.SOB-файлов** программы **Caligari trueSpace**, а также переделал функции, **выполняющие** инициализацию DirectDraw и переключение страниц, что связано с аномалиями работы альфа-смешивания и чтения из вторичного буфера DirectDraw. Как обычно, я хочу, чтобы вы бегло просмотрели заголовочный файл, просто чтобы понять, что в нем есть. Не волнуйтесь, если не сможете всего понять, главное — уловить идею.

## Структуры данных заголовочного файла

В приведенном далее тексте описывается заголовочный файл T3DLIB10.H. Мы подробно рассмотрим содержащиеся в нем функции и структуры (как вы знаете, я люблю начинать новую тему с обзора). Начнем с директив **#define**. Здесь не содержится ничего, кроме новых констант с фиксированной точкой и набора опций **визуализации**, которые нам понадобятся позднее, при написании единой **обобщающей** функции визуализации.

```
// Макроопределения //////////////////////////////////////
// Директива define альфа-смешивания

#define NUM_ALPHA_LEVELS 8 // Общее число альфа-уровней

// Директивы 1/z-буфера и перспективы
#define FIXP28_SHIFT 28 // Используется для
                        // 1/z-буферизации
#define FIXP22_SHIFT 22 // Используется для отображения
                        // текстуры с u/z- и
                        // v/z-перспективой

// Новые атрибуты для поддержки множественного отображения
// Многоугольник с множественным отображением
#define POLY4DV2_ATTR_MIPMAP 0x0400

// Объект с множественным отображением
#define OBJECT4DV2_ATTR_MIPMAP 0x0008

////////////////////////////////////
// Макроопределения для контроля атрибутов состояния функции
// визуализации. Заметим, что каждый класс управляющих
// флагов заключается в 4-битный слог. Это облегчает
// последующие расширения

// Z-буфер отсутствует. Визуализация многоугольников
// выполняется по списку
#define RENDER_ATTR_NOBUFFER 0x00000001

// Растеризация с использованием Z-буфера
#define RENDER_ATTR_ZBUFFER 0x00000002

// Растеризация с использованием I/z-буфера
#define RENDER_ATTR_INVZBUFFER 0x00000004

// Использование множественного отображения
#define RENDER_ATTR_MIPMAP 0x00000010

// Разрешение альфа-смешивания и перезаписи
#define RENDER_ATTR_ALPHA 0x00000020

// Разрешение билинейной фильтрации, но только в случае
// аффинных текстур или текстур с постоянным затенением
#define RENDER_ATTR_BILERP 0x00000040
```

```

// Использование аффинного текстурирования для всех
// многоугольников
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE 0x00000100

// Использование текстурирования с корректной перспективой
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT 0x00000200

// Использование текстурирования с линейно-кусочной
// перспективой
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR 0x00000400

// Использование гибридного текстурирования, сочетающего
// аффинное текстурирование с линейно-кусочной перспективой,
// зависящей от расстояния
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1 0x00000800

// Еще не реализовано
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID2 0x00001000

```

Затем идет новый тип, который я называю *контекстом визуализации*. В целом он позволяет инициализировать контекст необходимыми опциями визуализации. После этого вместо вызовов многочисленных функций визуализации используется вызов единой функции. Ниже представлена эта новая структура.

```

// Это новый тип, содержащий контекст визуализации, который
// позволяет не передавать функциям визуализации новые
// переменные, обходясь единой структурой
typedef struct RENDERCONTEXTV1_TYP
{
    int attr; // Атрибуты визуализации

    // Указатель на список визуализации
    RENDERLIST4DV2_PTR rend_list;
    UCHAR *video_buffer; // Указатель на видеобуфер
                        // визуализации
    int lpitch; // Шаг памяти видеобуфера (байт)

    UCHAR *zbuffer; // Указатель на Z-буфер или
                  // 1/z-буфер
    int zpitch; // Шаг памяти z- или 1/z-буфера в
              // байтах
    int alpha_override; // Значение альфа-параметра для
                      // всех многоугольников

    int mip_dist; // Максимальное расстояние, на
                 // котором нужно выполнить
                 // разделение на уровни
                 // множественного отображения
                 // 0 - (NUM_ALPHA_LEVELS - 1)
    int texture_dist; // Расстояние, при котором
                    // разрешается аффинное
                    // текстурирование при
                    // использовании гибридного режима
                    // текстурирования

```

```

int ival1, ival2;    // Дополнительные целочисленные
                    // параметры
float fval1, fval2; // Дополнительные параметры с
                    // плавающей точкой
void *vptr;         // Дополнительный указатель
} RENDERCONTEXTV1, *RENDERCONTEXTV1_PTR;

```

Большинство полей требуют внимательного рассмотрения. Среди них много новых полей, обеспечивающих множественное отображение и различные методы оптимизированного текстурирования. Мы опишем их при рассмотрении соответствующих тем.

А пока перейдем к глобальным переменным. Вводится только одна новая глобальная переменная с таблицей поиска, которая должна помочь нам в реализации альфа-смешивания.

```

// Эта таблица содержит все возможные RGB-значения,
// умноженные на некоторый скаляр
USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536];

```

И наконец, я приведу прототипы функций. Вы сможете увидеть все прототипы, с которыми мы будем иметь дело, и получить представление о том, что мы намерены делать в этой главе:

```

// ПРОТОТИПЫ //////////////////////////////////////

```

```

// Версии для Z-буфера

```

```

void Draw_Textured_TriangleZB2_16(
    POLYF4DV2_PTR face,    // Указатель на треугольник
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,         // Указатель на Z-буфер
    int zpitch);            // Байтов в строке Z-буфера

```

```

void Draw_Textured_Bilerp_TriangleZB_16(
    POLYF4DV2_PTR face,    // Указатель на треугольник
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,         // Указатель на Z-буфер
    int zpitch);            // Байтов в строке Z-буфера

```

```

void Draw_Textured_TriangleFSZB2_16(
    POLYF4DV2_PTR face,    // Указатель на треугольник
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,         // Указатель на Z-буфер
    int zpitch);            // Байтов в строке Z-буфера

```

```

void Draw_Textured_TriangleGSZB_16(
    POLYF4DV2_PTR face,    // Указатель на треугольник
    UCHAR *_dest_buffer,    // Указатель на видеобuffer
    int mem_pitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,         // Указатель на Z-буфер
    int zpitch);            // Байтов в строке Z-буфера

```

```

void Draw_Triangle_2DZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Gouraud_TriangleZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_RENDERLIST4DV2_SolidZB2_16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer, // Указатель на видеобuffer
    int lpitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера

// 1/z-версии

void Draw_Textured_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Textured_Bilerp_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Textured_TriangleFSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Textured_TriangleGSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Triangle_2DINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer

```

```

int mem_pitch,    // Байтов в строке, 320, 640 и т.д.
UCHAR *zbuffer,   // Указатель на Z-буфер
int zpitch);      // Байтов в строке Z-буфера

void Draw_Gouraud_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера

void Draw_Textured_Perspective_Triangle_FSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера

void Draw_Textured_PerspectiveLP_Triangle_INVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера

void Draw_RENDERLIST4DV2_SolidINVZB_16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer,
    int lpitch,
    UCHAR *zbuffer,
    int zpitch);

// Версии для Z-буфера и альфа-смешивания

void Draw_Textured_TriangleZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch,          // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_TriangleFSZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch,          // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_TriangleGSZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.

```

```

    UCHAR *zbuffer, // Указатель на Z-буфер
    int zpitch,     // Байтов в строке Z-буфера
    int alpha);

void Draw_Triangle_2DZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Gouraud_TriangleZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_RENDERLIST4DV2_SolidZB_Alpha16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer,
    int lpitch,
    UCHAR *zbuffer,
    int zpitch,
    int alpha_override);

// Версии для 1/z-буфера и альфа-смешивания
void Draw_Textured_TriangleINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_TriangleFSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_TriangleGSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

```

```

void Draw_Triangle_2DINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Gouraud_TriangleINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_Perspective_Triangle_INVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_PerspectiveLP_Triangle_INVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_Perspective_Triangle_FSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);

void Draw_RENDERLIST4DV2_SolidINVZB_Alpha16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer,
    int lpitch,
    UCHAR *zbuffer,
    int zpitch,
    int alpha_override);

```

// Версии без использования Z-буфера

```
void Draw_Textured_Triangle2_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_Bilerp_Triangle_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_TriangleFS2_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Triangle_2D3_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Gouraud_Triangle2_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_TriangleGS_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_Perspective_Triangle_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_PerspectiveLP_Triangle_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_Perspective_Triangle_FS_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_Textured_PerspectiveLP_Triangle_FS_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch); // Байтов в строке, 320, 640 и т.д.
```

```
void Draw_RENDERLIST4DV2_Solid2_16(  
    RENDERLIST4DV2_PTR rend_list,  
    UCHAR *_video_buffer, // Указатель на видеобuffer  
    int lpitch); // Байтов в строке, 320, 640 и т.д.
```

```

// Версии для альфа-смешивания без Z-буфера
int RGB_Alpha_Table_Builder(
    int num_alpha_levels, // Число создаваемых уровней
    // Таблица поиска
    USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536]);

void Draw_Triangle_2D_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке 320, 640 и т.д.
    int alpha);

void Draw_Textured_Triangle_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке 320, 640 и т.д.
    int alpha);

void Draw_Textured_TriangleFS_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке 320, 640 и т.д.
    int alpha);

void Draw_Textured_TriangleGS_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке 320, 640 и т.д.
    int alpha);

void Draw_Gouraud_Triangle_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке 320, 640 и т.д.
    int alpha);

void Draw_RENDERLIST4DV2_Solid_Alpha16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *_video_buffer, // Указатель на видеобuffer
    int lpitch,          // Байтов в строке 320, 640 и т.д.
    int alpha_override); // alpha value

// Новая функция визуализации работает во всех ситуациях!

void Draw_RENDERLISTV2_RENDERCONTEXTV1_16(
    RENDERCONTEXTV1_PTR rc);

// Новые функции моделей

int Load_OBJECT4DV2_COB2(
    OBJECT4DV2_PTR obj, // Указатель на объект
    char *filename,     // Имя Caligari COB-файла
    VECTOR4D_PTR scale, // Начальное масштабирование
    VECTOR4D_PTR pos,   // Начальное положение
    VECTOR4D_PTR rot    // Начальное направление вращения
    int vertex_flags,    // Флаги переупорядочения вершин

```

```

        // и выполнения преобразований
int mipmap=0); // Флаг множественного отображения
        // 1- множественное отображение,
        // 0 - его отсутствие

int Generate_Terrain2_OBJECT4DV2(
OBJECT4DV2_PTR obj, // Указатель на объект
float twidth, // Ширина в мировых координатах по
        // оси x
float theight // Высота (длина) в мировых
        // координатах по оси z
float vscale, // Вертикальный масштаб местности
char *height_map_file, //Имя файла битовой карты высот,
        // закодированных 256-цветной
        // палитрой
char *texture_map_file, //Имя файла карты текстур
int rgbcolor, // Цвет местности при отсутствии
        // текстур
VECTOR4D_PTR pos, // Начальное положение
VECTOR4D_PTR rot // Начальное направление вращения
int poly_attr, // Атрибуты затенения
float sea_level=-1, // Высота уровня моря
int alpha=-1); // Уровень альфа многоугольников
        // моря

// Новые функции DirectDraw

int DDraw_Init2(int width, int height int bpp,
        int windowed, int backbuffer_enable - 1);
int DDraw_Flip2(void);

// Функции множественного отображения
int Generate_Mipmaps(BITMAP_IMAGE_PTR source,
        BITMAP_IMAGE_PTR *mipmaps,
        float gamma = 1.01);
int Delete_Mipmaps(BITMAP_IMAGE_PTR *mipmaps,
        int leave_level_0);

```

Добро пожаловать в мой ночной кошмар! Как вам эти несметные **полчища** прототипов? А теперь представьте себе код, который использует их все...

## Построение нового базового растеризатора

Итак, начнем с самого начала. Первое, что нам нужно сделать, — это переписать базовые **растеризаторы**. Мы используем набор **растеризаторов**, что приводит к ряду проблем. Основную проблему вы могли заметить самостоятельно. При написании функции визуализации мы всегда округляли значения параметров вершин до целых **чисел**, а не оставляли их в виде чисел с плавающей точкой, которые затем более точно преобразовывались бы в самих растеризаторах в числа с фиксированной точкой, т.е. мы теряем точность. Напомню, что делает типичная **функция** визуализации с параметрами вершин перед тем, как передать их в функцию треугольника.

```

// Определяем вершины
face.tvlist[0].x =
    (int)rend_list->poly_ptrs[poly]->tvlist[0].x;

```

```
face.tvlist[0].y =  
    (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
```

Ранее нам было необходимо выполнение округления, поскольку первоначально растеризатор, работавший с числами с фиксированной точкой, требовал целочисленных координат. Сейчас проблема решена — теперь все функции, **работающие** с треугольниками, могут непосредственно получать числа с **плавающей** точкой, а затем округлять и преобразовывать их в числа с фиксированной точкой.

## Работа с числами с фиксированной точкой

Я окончательно решил работать с числами с фиксированной точкой, несмотря на то, что зачастую операции умножения и деления с числами с плавающей точкой **выполняются** ничуть не медленнее. Проблема в том, что **растеризация** представляет собой операцию интенсивного манипулирования с битами: тут идет работа и со значениями RGB-палитры, и с текстурными координатами, и с памятью. В этом случае числа с плавающей точкой — это просто сплошной тормоз: преобразование в целочисленный тип и из целочисленного типа в цикле растеризации происходит убийственно долго. Преобразование числа с плавающей точкой в целое число длится в худшем случае 70 тактов процессора, что совершенно неприемлемо. Кроме того, использование 32-битовых целых чисел позволяет выполнять все виды оптимизации, что невозможно в случае чисел с плавающей точкой.

Итак, мы будем использовать числа с плавающей точкой для всех задач, кроме растеризации. В случае расчета преобразований, освещенности и т.п. плавающая точка позволяет работать быстрее и точнее, однако при выполнении растеризации она дает убийственно низкую производительность. Я экспериментировал с плавающей точкой — написал версии кода с использованием **плавающей** точки, фиксированной точки, а также гибридную версию, и пришел к выводу, что 32-битные числа с фиксированной точкой обеспечивают гораздо большую скорость растеризации.

Тем не менее, для обеспечения большей точности десятичных вычислений старый формат чисел с фиксированной точкой 16.16 должен быть дополнен другими форматами, такими как 4.28 и 10.22. Я рассмотрю эти новые форматы там, где они будут нужны — в разделах, посвященных использованию 1/z-буфера и текстур с корректной перспективой.

Таким образом, мы собираемся использовать вычисления с фиксированной точкой на протяжении всего периода работы с **растеризаторами**. Я также пересмотрел базовые **растеризаторы** с тем, чтобы они лучше соответствовали соглашению о порядке заполнения с верхнего левого угла. Их точность поднята до субпиксельного уровня, благодаря чему **текстурирование** и визуализация сплошных тел выполняются очень чисто. Неверные соглашения о заполнении приводят к наличию в сцене разрывов и перерисовки пикселей. Учитывая все сказанное, давайте рассмотрим новые базовые растеризаторы, построенные на этих новых свойствах.

## Новые растеризаторы без Z-буферизации

Вначале рассмотрим базовый **растеризатор** без Z-буфера. Конечно, есть новые растеризаторы с корректной перспективой, альфа-смешиванием и множественным отображением. Но мы не будем забегать вперед и рассмотрим только переписанные версии **растеризаторов**, которые у нас уже есть.

### Функция

```
void Draw_Triangle_2D3_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник
```

```
UCHAR*_dest_buffer, // Указатель на видеобуфер
int mem_pitch); // Байт в строке, 320, 640 и т.д.
```

#### Описание

Функция `Draw_Triangle_2D3_16()` просто выводит на экран многоугольник с постоянным или плоским затенением. Конечно, здесь отсутствует **Z-буферизация** и многоугольник будет нарисован без какого-либо учета **порядка** его визуализации. Работа по предварительной сортировке многоугольников возлагается на **вызывающую** функцию. Как видите, у функции весьма запоминающееся имя, но это, пожалуй, единственное отличие от предыдущей версии 2.0. Однако сильной стороной функции является то, что она работает быстрее, чище и имеет точность субпиксельного **уровня**, **поддерживая** соглашение о заполнении с верхнего левого угла. Кроме того, теперь данной функции, как и всем другим **растеризаторам**, передаются не отдельные координаты вершин треугольника, а единый указатель.

#### Функция

```
void Draw_Gouraud_Triangle2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR*_dest_buffer, // Указатель на видеобуфер
    int mem_pitch); // Байт в строке, 320, 640 и т.д.
```

#### Описание

Функция `Draw_Gouraud_Triangle2_16()` — это улучшенная версия предыдущей функции, работающая с субпиксельной точностью и более строгим соглашением о заполнении с левого верхнего угла.

#### Функция

```
void Draw_Textured_Triangle2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR*_dest_buffer, // Указатель на видеобуфер
    int mem_pitch); // Байт в строке, 320, 640 и т.д.
```

#### Описание

Функция `Draw_Textured_Triangle2_16()` — это улучшенная версия предыдущей функции с аффинным текстурным отображением, работающая с субпиксельной точностью и более строгим соглашением о заполнении с левого верхнего угла. Эта функция улучшает **текстурирование** и делает его более сплошным.

НА ЗАМЕТКУ

По правде говоря, аффинное текстурирование **выполняется** настолько красиво и быстро, что мне даже не хочется использовать перспективу.

#### Функция

```
void Draw_Textured_TriangleFS2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR*_dest_buffer, // Указатель на видеобуфер
    int mem_pitch); // Байт в строке, 320, 640 и т.д.
```

#### Описание

Функция `Draw_Textured_TriangleFS2_16()` — это улучшенная версия предыдущей функции аффинного текстурного отображения, **поддерживающая** плоское затенение, с субпиксельной точностью и более строгим соглашением о заполнении с левого верхнего угла. Эта функция улучшает текстурирование и делает его более сплошным.

#### Функция

```
void Draw_RENDERLIST4DV2_Solid2_16(
    RENDERLIST4DV2_PTR rend_list, // Список визуализации
```

```

    UCHAR *_dest_buffer,    // Указатель на видеобуфер
    int mem_pitch);         // Байт в строке

```

### Описание

Функция `Draw_RENDERLIST4DV2_Solid2_16()` — это входная точка для вызова перечисленных ранее функций, т.е. функций без использования Z-буфера. Внутреннее устройство функции идентично предыдущей версии, однако удалено округление до целых чисел. Ниже для справки приводится листинг функции.

```

void Draw_RENDERLIST4DV2_Solid2_16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *_video_buffer,
    int lpitch)
{
    // 16-битная версия
    // Эта функция "обслуживает" список визуализации, иными
    // словами, выводит все грани, перечисленные в
    // списке. Функция вызывает соответствующий
    // растеризатор, основанный на модели освещенности
    // многоугольников

    POLYF4DV2 face;    // Временная грань, используемая
                       // для визуализации многоугольника

    // Все, что у нас есть, это список многоугольников,
    // которые будут выведены
    for(int poly=0; poly < rend_list->num_polys; poly++)
    {
        // Визуализация многоугольника производится тогда и
        // только тогда, когда он не отсечен, не отбракован,
        // активен и видим. Заметим, что концепция обратной
        // поверхности в случае каркасного процессора не
        // применима
        if (!(rend_list->poly_ptrs[poly]->state &
            POLY4DV2_STATE_ACTIVE) ||
            (rend_list->poly_ptrs[poly]->state &
            POLY4DV2_STATE_CLIPPED) ||
            (rend_list->poly_ptrs[poly]->state &
            POLY4DV2_STATE_BACKFACE) )
            continue; // Переход к следующему многоугольнику
        // Вначале необходимо проверить текстуру
        // многоугольника, поскольку она может быть либо
        // излучающей, либо с плоским затенением, поэтому
        // нужно будет вызывать различные растеризаторы
        if (rend_list->poly_ptrs[poly]->attr &
            POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
        {
            // Устанавливаем вершины
            face.tvlist[0].x = (float)rend_list->
                poly_ptrs[poly]->tvlist[0].x;
            face.tvlist[0].y = (float)rend_list->
                poly_ptrs[poly]->tvlist[0].y;
            face.tvlist[0].z = (float)rend_list->
                poly_ptrs[poly]->tvlist[0].z;

```

```

face.tvlist[0].u0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[0].u0;
face.tvlist[0].v0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[0].v0;

face.tvlist[1].x - (float)rend_list->
    poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y - (float)rend_list->
    poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z - (float)rend_list->
    poly_ptrs[poly]->tvlist[1].z;
face.tvlist[1].u0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[1].u0;
face.tvlist[1].v0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[1].v0;

face.tvlist[2].x - (float)rend_list->
    poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y - (float)rend_list->
    poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z - (float)rend_list->
    poly_ptrs[poly]->tvlist[2].z;
face.tvlist[2].u0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[2].u0;
face.tvlist[2].v0 - (float)rend_list->
    poly_ptrs[poly]->tvlist[2].v0;

// Присваиваем текстуру
face.texture - rend_list->
    poly_ptrs[poly]->texture;
// Является ли эта текстура простой излучающей
// текстурой?
if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
{
    // Выводим текстурированный треугольник как
    // излучающий
    Draw_Textured_Triangle2_6(&face,
        video_buffer,
        lpitch);
    // Draw_Textured_Perspective_Triangle_16(
    //     &face, video_buffer, lpitch);
} // if
else if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_FLAT)
{
    // Выводим как текстуру с плоским затенением
    face.lit_color[0] = rend_list->
        poly_ptrs[poly]->lit_color[0];
    Draw_Textured_TriangleFS2_16(&face,
        video_buffer,
        lpitch);
    // Draw_Textured_Perspective_Triangle_FS_16(
    //     &face, video_buffer, lpitch);
}

```

```

} // else
else
{
    // Должен быть флаг
    // POLY4DV2_ATTR_SHADE_MODE_GOURAUD
    face.lit_color[0] = rend_list->
        poly_ptrs[poly]->lit_color[0];
    face.lit_color[1] = rend_list->
        poly_ptrs[poly]->lit_color[1];
    face.lit_color[2] = rend_list->
        poly_ptrs[poly]->lit_color[2];
    Draw_Textured_TriangleGS_16(&face,
        video_buffer,
        lpitch);
} // else
} // if
else if ((rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
    (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT))
{
    // Выводим как текстуру с постоянным затенением
    face.lit_color[0] = rend_list->
        poly_ptrs[poly]->lit_color[0];

    // Устанавливаем вершины
    face.tvlist[0].x = (float)rend_list->
        poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rend_list->
        poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z = (float)rend_list->
        poly_ptrs[poly]->tvlist[0].z;

    face.tvlist[1].x = (float)rend_list->
        poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y = (float)rend_list->
        poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z = (float)rend_list->
        poly_ptrs[poly]->tvlist[1].z;

    face.tvlist[2].x = (float)rend_list->
        poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y = (float)rend_list->
        poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z = (float)rend_list->
        poly_ptrs[poly]->tvlist[2].z;

    // Выводим треугольник с помощью основного
    // плоского растеризатора
    Draw_Triangle_2D3_16(&face, video_buffer,
        lpitch);
} // if
else if (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_GOURAUD)

```

```

// Указываем вершины
face.tvlist[0].x = (float)rend_list->
    poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y = (float)rend_list->
    poly_ptrs[poly]->tvlist[0].y;
face.tvlist[0].z = (float)rend_list->
    poly_ptrs[poly]->tvlist[0].z;
face.lit_color[0] = rend_list->
    poly_ptrs[poly]->lit_color[0];

face.tvlist[1].x = (float)rend_list->
    poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y = (float)rend_list->
    poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z = (float)rend_list->
    poly_ptrs[poly]->tvlist[1].z;
face.lit_color[1] = rend_list->
    poly_ptrs[poly]->lit_color[1];

face.tvlist[2].x = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].z;
face.lit_color[2] = rend_list->
    poly_ptrs[poly]->lit_color[2];

// Выводим треугольник с затенением по Гуро
Draw_Gouraud_Triangle2_16(&face, video_buffer,
    lpitch);
} // if gouraud
} // poly
} // Draw_RENDERLIST4DV2_Solid2_16

```

Обратите внимание на все приведения к типу float — все вершины обрабатываются в исходном виде, без потери точности. Теперь давайте займемся растеризаторами с Z-буфером, отложив на время в сторону новые технологии, такие как альфа-смешивание.

## Новые растеризаторы с Z-буфером

Ниже перечислены новые базовые функции для Z-буферизации. Список функций не является исчерпывающим, он не включает альфа-смешивание, поскольку мы еще не рассматривали эту тему. Этими функциями замещаются уже использовавшиеся нами ранее функции.

### Функция

```

void Draw_Triangle_2DZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байт в строке буфера

```

### Описание

Функция `Draw_Triangle_2DZB2_16()` — основная новая функция с Z-буферизацией; она просто выводит на экран Z-буферизованный треугольник и обновляет Z-буфер при выполнении данной операции. Функция работает с субпиксельной точностью в соответствии с соглашением о заполнении с левого верхнего угла. Параметры функции не изменились, а изменилось только ее имя.

### Функция

```
void Draw_Textured_TriangleZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);          // Байт в строке буфера
```

### Описание

Функция `Draw_Textured_TriangleZB2_16()` — это новая функция с Z-буферизацией, которая выводит треугольник с аффинным текстурированием и постоянным затенением. За счет субпиксельной точности и выполнения соглашения о заполнении с левого верхнего угла эта функция работает значительно лучше. Соглашения о вызове те же, что и ранее.

### Функция

```
void Draw_Textured_TriangleFSZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);          // Байт в строке буфера
```

### Описание

Функция `Draw_Textured_TriangleFSZB2_16()` — это новая функция с Z-буферизацией, которая выводит треугольник с аффинным текстурированием и плоским затенением. За счет субпиксельной точности и выполнения соглашения о заполнении с левого верхнего угла эта функция работает значительно лучше. Соглашения о вызове те же, что и ранее.

### Функция

```
void Draw_Gouraud_TriangleZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);          // Байт в строке буфера
```

### Описание

Функция `Draw_Gouraud_TriangleZB2_16()` — это новая функция с Z-буферизацией, которая выводит треугольник с затенением по Гуро. За счет субпиксельной точности и выполнения соглашения о заполнении с левого верхнего угла эта функция работает значительно лучше. Соглашения о вызове те же, что и ранее.

### Функция

```
void Draw_RENDERLIST4DV2_SolidZB2_16(
    RENDERLIST4DV2_PTR rend_list,
    UCHAR *video_buffer,
    int lpitch,
```

```
UCHAR *zbuffer,
int zpitch);
```

#### Описание

Функция `Draw_RENDERLIST4DV2_SolidZB2_16()` — это входная точка для вызова перечисленных ранее функций, т.е. функция с Z-буферизацией. Устройство функции идентично предыдущей версии, однако здесь отсутствует округление до целых чисел.

Примерно на этом месте мы остановились в предыдущей главе. Мы могли рисовать сплошные многоугольники с постоянным, плоским затенением или затенением по Гуро, а также многоугольники с аффинной текстурой, имеющие постоянное или плоское затенение. Описанные выше функции делают примерно то же самое, но чище, точнее, с меньшим числом ошибок. В демонстрационной программе нет никакого смысла; она выглядит так же, как и раньше. Давайте теперь перейдем к тому, в чем есть заметные отличия.

## Текстурирование с затенением по Гуро

В первом издании библиотеки **растеризаторов** существует поддержка текстур с затенением по Гуро. Хотя я вначале думал, что такая программа будет работать слишком медленно, после экспериментов с ней я обнаружил, что она работает вполне удовлетворительно. Наш небольшой **растеризатор** с фиксированной точкой действительно быстрый, так что хотя внутренний цикл и более сложный в связи с поддержкой затенения по Гуро, поскольку мне пришлось добавить несколько **интерполянтов**, это дает совсем **небольшое** замедление. В любом случае давайте поговорим о том, что необходимо для **поддержки** затенения по Гуро. На рис. 12.1 показаны две схемы аффинного **текстурирования** с постоянным затенением и одноцветным затенением по Гуро. Фокус состоит в том, чтобы произвести их слияние.

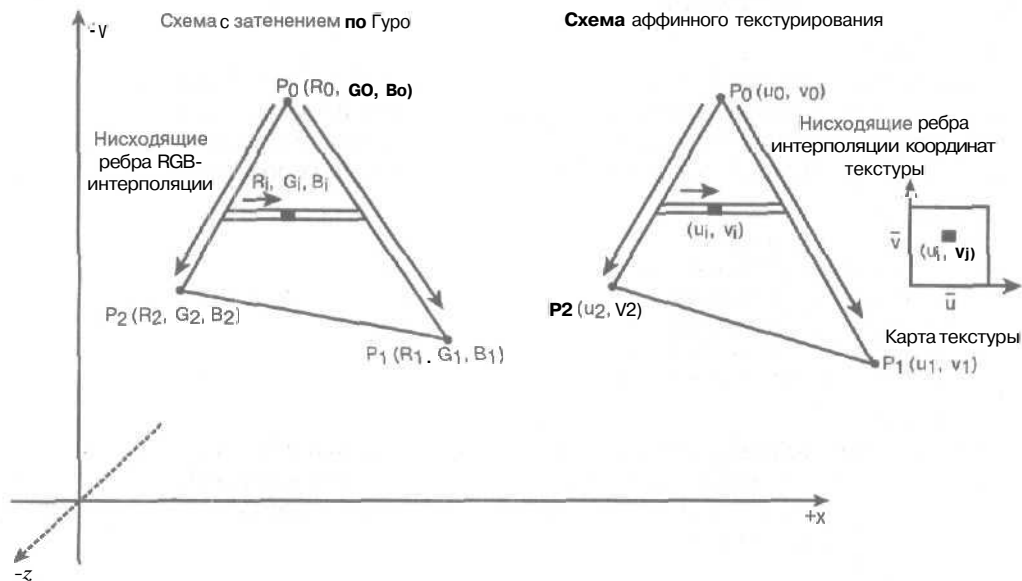


Рис. 12.1. Слияние схем многоугольников текстурирования с **постоянным** затенением и **затенением по Гуро**

Мой план атаки прост: мы начнем с функции затенения по Гуро, которая рисует одноцветный многоугольник, добавляем поддержку интерполяции координат  $u$  и  $v$  для **тек-**

стурирования, а затем объединяем их, используя функцию затенения по Гуро как основу (поскольку она представляет большую из двух функций, мы добавляем к ней меньшую). Давайте посмотрим, что нам нужно сделать, начиная с кода затенения по Гуро. Ниже приводится фрагмент внутреннего цикла, который рисует многоугольник (версия без Z-буферизации),

```
// Устанавливаем указатель экрана на начальную строку
screen_ptr = dest_buffer + (ystart * mem_pitch);

for (yi = ystart; yi < yend; yi++)
{
    // Вычисляем конечные точки
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные точки для интерполянтов u, v, w
    ui = ul + FIXP16_ROUND_UP;
    vi = vl + FIXP16_ROUND_UP;
    wi = wl + FIXP16_ROUND_UP;

    // Вычисляем интерполянты u, v, w
    if ((dx = (xend - xstart)) > 0)
    {
        du = (ur - ul)/dx;
        dv = (vr - vl)/dx;
        dw = (wr - wl)/dx;
    } // if
    else
    {
        du = (ur - ul);
        dv = (vr - vl);
        dw = (wr - wl);
    } // else

    // Вывод на экран
    for (xi = xstart; xi < xend; xi++)
    {
        // Пишем текстель в формате 5,6,5
        screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
            ((vi >> (FIXP16_SHIFT+2)) << 5) +
            (wi >> (FIXP16_SHIFT+3));
        // Интерполируем u, v, w
        ui+=du;
        vi+=dv;
        wi+=dw;
    } // for xi

    // Интерполируем u, v, w, x
    // вдоль правого и левого ребер
    xl+=dxdyl;
    ul+=dudyl;
    vl+=dvdyL;
    wl+=dwdyl;
```

```

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;

// Продвигаем указатель экрана
screen_ptr+=mem_pitch;
} // for y

```

Я выделил внутренний цикл, выводящий одиночную строку, основанный на интерполяции красного, зеленого и синего цветов в формате 16.16, хранящихся в параметрах *u*, *v* и *w*, соответственно. Этот короткий отрезок кода строит 16-битное RGB-слово в формате 5.6.5 из значений с фиксированной точкой и выводит его на экран.

```

screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
                 ((vi >> (FIXP16_SHIFT+2)) << 5) +
                 (wi >> (FIXP16_SHIFT+3));

```

Ничего другого в функции затенения по Гуро нет. Теперь давайте посмотрим на **растеризатор** аффинной текстуры, чтобы понять, что он **интерполирует**.

```

// Устанавливаем указатель экрана на начальную строку
screen_ptr = dest_buffer + (ystart * mem_pitch);

```

```

for (yi = ystart; yi < yend; yi++)
{
    // Вычисляем конечные точки
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные точки для интерполянтов u, v
    ui = ul + FIXP16_ROUND_UP;
    vi = vl + FIXP16_ROUND_UP;

    // Вычисляем интерполянты u, v
    if ((dx = (xend - xstart)) > 0)
    {
        du = (ur - ul) / dx;
        dv = (vr - vl) / dx;
    } // if
    else
    {
        du = (ur - ul);
        dv = (vr - vl);
    } // else

    // Вывод на экран
    for (xi = xstart; xi < xend; xi++)
    {
        // Выводим текстур
        screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
                                   ((vi >> FIXP16_SHIFT) << texture_shift2)];
        // Интерполируем u, v, w
        ui += du;
        vi += dv;
    }
}

```

```

} // for xi

// Интерполируем u, v, w, x вдоль
// правого и левого ребер
xl+=dxdyl;
ul+=dudyl;
vl+=dvdy;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;

// Продвигаем указатель экрана
screen_ptr+=mem_pitch;
} // for y

```

Опять реальный код для вывода пикселя довольно **тривиален**.  $u_i$  и  $v_i$  — это координаты текстуры с фиксированной точкой, которые преобразуются из формата 16.16 в целые числа, а затем используются для доступа к карте текстур. Затем **текстель** (пиксель текстуры) записывается в память — и это все. **Следовательно**, объединение этих двух подпрограмм не должно быть очень трудным. Необходимо помнить, что у нас используется затенение по Гуро, и, как и в случае с плоским затенением, предполагается, что цвет под текстурой белый. **Основываясь** на этом, мы выполнили вычисления освещенности с белым **базовым цветом** и затем использовали эти результаты для модуляции цвета текстуры. Здесь мы собираемся делать то же самое. Рисунок 12.2 показывает поток данных в этом процессе.

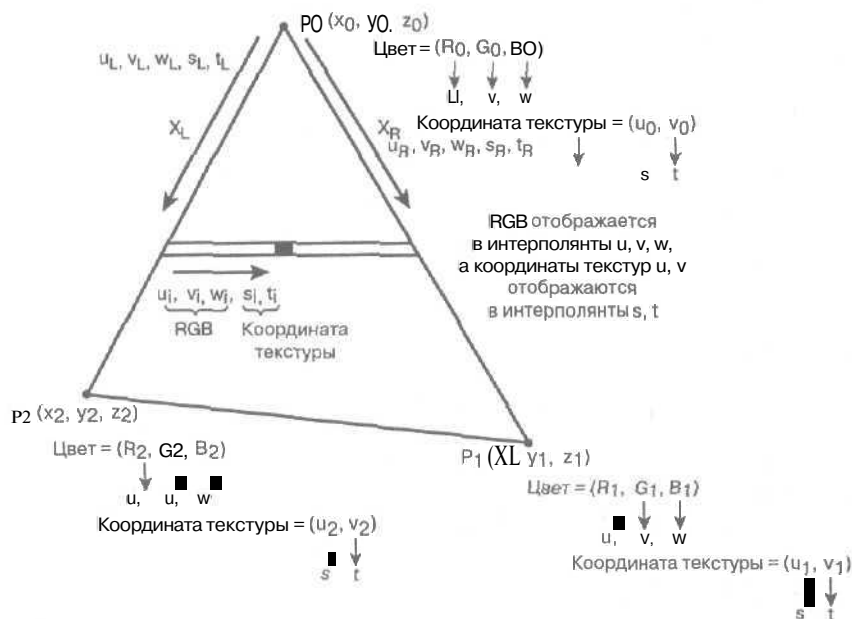


Рис. 12.2. Интерполяционный поток отображения текстуры по Гуро

Затенение по Гуро и отображение координат текстуры происходят одновременно. Затенение по Гуро интерполирует параметры  $r$ ,  $d$ , и  $B$  (с помощью **интерполянтов**  $u$ ,  $v$  и  $w$ ), тогда

как преобразователь координат текстуры интерполирует параметры  $u$  и  $v$  (с помощью интерполянтов  $s$  и  $t$ ). На последней стадии выполняется доступ к текстуре и модуляция цвета. Посмотрите на фрагмент кода преобразователя координат текстуры по Гуро.

```
// Устанавливаем указатель экрана на начальную строку
screen_ptr = dest_buffer + (ystart * mem_pitch);
for (yi = ystart; yi < yend; yi++)
{
    // Вычисляем конечные точки
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные точки для
    // интерполянтов u, v, w, s, t
    ui = ul + FIXP16_ROUND_UP;
    vi = vl + FIXP16_ROUND_UP;
    wi = wl + FIXP16_ROUND_UP;
    si = sl + FIXP16_ROUND_UP;
    ti = tl + FIXP16_ROUND_UP;

    // Вычисляем интерполянты u, v, w, s, t
    if ((dx - (xend - xstart)) > 0)
    {
        du = (ur - ul) / dx;
        dv = (vr - vl) / dx;
        dw = (wr - wl) / dx;
        ds = (sr - sl) / dx;
        dt = (tr - tl) / dx;
    } // if
    else
    {
        du = (ur - ul);
        dv = (vr - vl);
        dw = (wr - wl);
        ds = (sr - sl);
        dt = (tr - tl);
    } // else

    // Вывод на экран
    for (xi = xstart; xi < xend; xi++)
    {
        // Вначале получаем текстель
        textel = textmap[(si >> FIXP16_SHIFT) +
            ((ti >> FIXP16_SHIFT) << texture_shift2)];

        // Извлечение rgb-компонентов
        r_textel = ((textel >> 11));
        g_textel = ((textel >> 5) & 0x3f);
        b_textel = (textel & 0x1f);

        // Модуляция текстеля с затенением по Гуро
        r_textel *= ui;
```

```

g_textel*=vi;
b_textel*=wi;

// Окончательная запись текста. Заметим, что мы
// выполнили математические операции, в результате
// которых получены значения r*32, g*64, b*32.
// Следовательно, нам нужно разделить результаты на
// 32, 64 и 32, соответственно. Но поскольку нам
// нужно выполнить сдвиг результатов, чтобы можно
// было записать их в формате 5.6.5, мы можем
// воспользоваться преимуществами битовых сдвигов
screen_ptr[xi] - ((b_textel >> (FIXP16_SHIFT+8)) +
    (g_textel >> (FIXP16_SHIFT+8)) << 5) +
    (r_textel >> (FIXP16_SHIFT+8)) << 11));

// Интерполируем u, v, w, s, t
ui+=du;
vi+=dv;
wi+=dw;
si+=ds;
ti+=dt;
} // for xi

// Интерполируем u, v, w, x, s, t
// вдоль правого и левого ребер
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
wl+=dwdyl;
sl+=dsdyl;
tl+=dtdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;
sr+=dsdyr;
tr+=dtdyr;

// Продвигаем указатель экрана
screen_ptr+=mem_pitch;
} // for y

```

Как обычно, я выделил внутренний цикл. Первый шаг состоит в извлечении текста, который нужно подвергнуть затенению по Гуро. Это реализуется с помощью следующего кода.

```

r_textel = ((textel >> 11) );
g_textel = ((textel >> 5) & 0x3f);
b_textel = (textel & 0x1f);

```

Здесь r\_textel, g\_textel и b\_textel содержат RGB-значения текста в формате 5.6.5. Нам нужно умножить эти значения на значения Гуро-интерполянтов красного, зеленого и синего цветов, интерполирующих "белый" фантомный многоугольник под текстурой. Ниже представлен код этой операции.

```

r_textel*=ui;
g_textel*=vi;
b_textel*=wi;

```

Давайте немного прервемся, чтобы посмотреть, что происходит дальше с математикой с фиксированной точкой. `r_textel`, `g_textel` и `b_textel` более не являются числами с фиксированной точкой — они стали целыми числами в диапазонах (0..31), (0..63) и (0..31), соответственно. Однако `ui`, `vi` и `wi` — это числа с фиксированной точкой. Поэтому, умножая число с фиксированной точкой на целое число, мы получим число с фиксированной точкой. Следовательно, `r_textel`, `g_textel` и `b_textel` представляют собой значения `r`, `g` и `b` параметров, модулированных затенением по Гуро. Нам необходимо извлечь из них правильные данные, поскольку они являются числами с фиксированной точкой с масштабированием. Затем их нужно сдвинуть, чтобы они попали в требуемый диапазон, и записать окончательный пиксель. Ниже приведен код, который для выполнения этой задачи манипулирует битами, приводя параметры `r`, `g` и `b` к формату 5.6.5.

```

screen_ptr[xi] - ((b_textel >> (FIXP16_SHIFT+8)) +
((g_textel >> (FIXP16_SHIFT+8)) << 5) +
((r_textel >> (FIXP16_SHIFT+8)) << 11));

```

#### НА ЗАМЕТКУ

Например, если к числу 10 в формате с фиксированной точкой прибавить число 10 в формате с фиксированной точкой, получится число 20 с фиксированной точкой. Это то же самое, что умножить число 10 с фиксированной точкой на 2. Давайте посмотрим почему. В формате 16.16  $10 = (10 * 65536)$ . Следовательно, если к числу 10 с фиксированной точкой прибавить число 10 с фиксированной точкой, получится  $(10 * 65536) + (10 * 65536) = (20 * 65536) = (2 * 10 * 65536) = 2 * (10 * 65536)$ .

Поэтому мы можем умножать или делить числа с фиксированной точкой на целые числа, и они всегда останутся числами с фиксированной точкой.

Итак, сейчас у нас есть очень быстро работающая полная программа растеризации с текстурным отображением, затенением по Гуро, которая содержит всего лишь три операции умножения, несколько операций сложения и сдвига. Именно поэтому мы собираемся использовать математические операции с фиксированной точкой — плавающая точка работает очень медленно за счет операций преобразования.

Если вы рассуждаете так же, как я, то, вероятно, попробуете придумать способы ускорить работу программы. Конечно, есть много способов, но алгоритмически код почти оптимален, что означает, что он выполняет почти минимальный объем работы, необходимый для выполнения поставленной задачи. Есть еще несколько операций битового масштабирования, которые можно было бы выполнить, но по большому счету, уже нет способов при меньшем объеме работы получить тот же результат.

Могут помочь таблицы поиска, но с ними связаны проблемы кэширования и я не уверен, что есть много ситуаций, в которых мы можем их использовать. Наиболее очевидный пример такой таблицы — предвычисленные значения всех возможных произведений текстелей, но это была бы очень большая таблица. Например, все возможные произведения двух RGB-значений в формате 5.6.5 ориентировочно представляют собой  $65\,536 \cdot 65\,536 = 2^{32} \approx 4$  млрд. значений. Увы, это слишком много! Другой способ — это использовать для расчетов уменьшенные значения; скажем, масштабировать каждое до 12 битов — тогда это будет примерно  $2^{12} \cdot 2^{12} = 2^{24} \approx 16.7$  млн. значений. Все равно это слишком много, а кроме того, мы сильно урезаем цветовую палитру. Я не говорю, что нельзя использовать таблицу поиска, я лишь утверждаю, что это нерационально. Все, что нужно сделать с нашим кодом, чтобы он работал быстрее, — это немного пере-

строить его и внести некоторые другие незначительные изменения, а структура его и так достаточно хороша.

Ниже приводится реальная функция, содержащая текстурный преобразователь с затенением по Гуро без Z-буферизации.

```
void Draw_Textured_TriangleGS_16(  
    POLYF4DV2_PTRface, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобуфер  
    int mem_pitch)      // Байт в строке, 320, 640 и т.д.
```

Вы можете вызывать функцию так же, как и в случае с версией для плоского затенения, но поскольку здесь используется затенение по Гуро и отображение **текстур**, вы должны указать как координаты текстуры *u*, *v*, так и цвета вершин, которые должны храниться в `lit_colors[0..2]` — помните массив цветов вершин? Кроме того, проанализируйте версию функции с Z-буфером.

```
void Draw_Textured_TriangleGSZB_16(  
    POLYF4DV2_PTRface, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобуфер  
    int mem_pitch)      // Байт в строке, 320, 640 и т.д.  
    UCHAR *_zbuffer,    // Указатель на Z-буфер  
    int zpitch);        // Байт в строке Z-буфера
```

Здесь также необходимо, чтобы передняя поверхность имела координаты текстуры и набор значений **освещенности** вершин, и наконец, нужно указать Z-буфер и шаг Z-буфера. Вот пример фрагмента кода функции визуализации, вызывающей функцию с **текстурированием** по Гуро.

**// Настраиваем список вершин и координаты текстуры**

```
face.tvlist[0].x = (float)rend_list->  
    poly_ptrs[poly]->tvlist[0].x;  
face.tvlist[0].y = (float)rend_list->  
    poly_ptrs[poly]->tvlist[0].y;  
face.tvlist[0].z = (float)rend_list->  
    poly_ptrs[poly]->tvlist[0].z;  
face.tvlist[0].u0 = (float)rend_list->  
    poly_ptrs[poly]->tvlist[0].u0;  
face.tvlist[0].v0 = (float)rend_list->  
    poly_ptrs[poly]->tvlist[0].v0;  
  
face.tvlist[1].x = (float)rend_list->  
    poly_ptrs[poly]->tvlist[1].x;  
face.tvlist[1].y = (float)rend_list->  
    poly_ptrs[poly]->tvlist[1].y;  
face.tvlist[1].z = (float)rend_list->  
    poly_ptrs[poly]->tvlist[1].z;  
face.tvlist[1].u0 = (float)rend_list->  
    poly_ptrs[poly]->tvlist[1].u0;  
face.tvlist[1].v0 = (float)rend_list->  
    poly_ptrs[poly]->tvlist[1].v0;  
  
face.tvlist[2].x = (float)rend_list->  
    poly_ptrs[poly]->tvlist[2].x;  
face.tvlist[2].y = (float)rend_list->  
    poly_ptrs[poly]->tvlist[2].y;
```

```

face.tvlist[2].z = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].z;
face.tvlist[2].u0 = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].u0;
face.tvlist[2].v0 = (float)rend_list->
    poly_ptrs[poly]->tvlist[2].v0;

// Присваиваем текстуру
face.texture = rend_list->poly_ptrs[poly]->texture;

// Настраиваем цвета вершин
face.lit_color[0] = rend_list->
    poly_ptrs[poly]->lit_color[0];
face.lit_color[1] = rend_list->
    poly_ptrs[poly]->lit_color[1];
face.lit_color[2] = rend_list->
    poly_ptrs[poly]->lit_color[2];

```

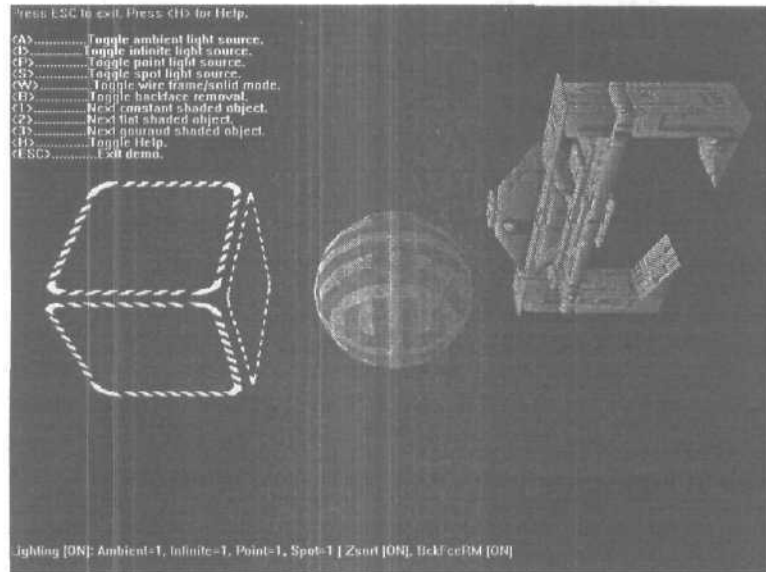


Рис. 12.3. Копия экрана работы текстурного растеризатора с затенением по Гуро

Это почти все, что касается **текстурирования** по Гуро. Мы уже не раз встречались с используемыми здесь методиками; так что теперь мы просто собрали известные нам методики в единую функцию.

В качестве демонстрационной программы текстурирования по Гуро на прилагаемом компакт-диске представлена программа **DEMO112\_1.CPP|EXE**. На рис. 12.3 показана копия экрана этой демонстрационной программы. Здесь вы видите группу объектов, имеющих постоянную, плоскую текстуру, а также текстуру по Гуро (слева направо), и можете увидеть, в чем их отличие. Чтобы выбрать конкретный объект, нажмите <1>, <2> или <3>. Обратите внимание, что **текстурирование** по Гуро не очень помогает в случае угловых объектов, однако для гладких объектов это именно то, что надо.

## Прозрачность и альфа-смешивание

По правде говоря, я немного беспокоился, что добавление прозрачности в процессор понизит производительность последнего. Теоретически альфа-смешивание представляет собой простую операцию смешивания. Если заданы два пикселя  $p\_source1(r,g,b)$  и  $p\_source2(r,g,b)$ , а также коэффициент смешивания  $\alpha$  (в пределах от 0 до 1), мы можем вычислить конечный пиксель  $p\_final(r,g,b)$ , который смешивает их. Формула данной операции выглядит следующим образом:

$$p\_final = (\alpha) * (p\_source1) + (1-\alpha) * (p\_source2);$$

Каждый пиксель состоит из трех компонент —  $r$ ,  $g$  и  $b$ . Следовательно, уравнение в действительности представляет собой целых три:

$$\begin{aligned} p\_finalr &= (\alpha) * (p\_source1r) + (1-\alpha) * (p\_source2r) \\ p\_finalg &= (\alpha) * (p\_source1g) + (1-\alpha) * (p\_source2g) \\ p\_finalb &= (\alpha) * (p\_source1b) + (1-\alpha) * (p\_source2b) \end{aligned}$$

Если  $\alpha=1$ ,  $p\_source1$  полностью непрозрачен, а  $p\_source2$  вообще не участвует в смешивании. Напротив, если  $\alpha=0$ ,  $p\_source1$  полностью прозрачен, а  $p\_source2$  полностью непрозрачен. Наконец, если  $\alpha=0.5$ , прозрачности  $p\_source1$  и  $p\_source2$  соотносятся как 50/50. Это основа нашей технологии альфа-смешивания. Итак, у нас есть три основные проблемы.

- Как выполнить эту сложную математическую операцию достаточно быстро?
- Как реализовать поддержку альфа-смешивания в процессоре? Т.е. как нам указать, что многоугольники содержат параметр  $\alpha$ ?
- Что нам делать с замедлением DirectX до 30 раз при чтении из вторичного буфера?

Рассмотрим первую проблему.

Проанализируем функцию, **содержащую** шесть операций умножения и три операции сложения на один пиксель. Конечно, мы можем применить математику **альфа-смешивания** к процедуре обработки каждого пикселя в цикле растеризации, но это неприемлемо. Есть лучший путь. Это как раз тот случай, когда нам могут помочь таблицы поиска.

## Использование таблиц поиска для альфа-смешивания

Наша проблема заключается в расчетах **альфа-смешивания**:

$$\begin{aligned} p\_finalr &= (\alpha) * (p\_source1r) + (1-\alpha) * (p\_source2r) \\ p\_finalg &= (\alpha) * (p\_source1g) + (1-\alpha) * (p\_source2g) \\ p\_finalb &= (\alpha) * (p\_source1b) + (1-\alpha) * (p\_source2b) \end{aligned}$$

Здесь есть не только шесть умножений, но еще и три сложения и одно дополнительное сложение для вычисления константы  $(1-\alpha)$ , т.е. в итоге четыре сложения. Тот факт, что это чистая математика, ни о чем не говорит, поскольку мы не можем выполнить эти вычисления с нашими пикселями, т.к. они даны в RGB-формате и их еще нужно извлечь из него. Поэтому нам нужно создать таблицу поиска, которая получает два RGB-значения в формате 5.6.5 и **альфа-значение** в диапазоне 0-255 и выдает результат смешивания. Графически это показано на рис. 12.4. И снова мы сталкиваемся с проблемой огромной таблицы поиска. Ниже приводится расчет ее размеров.

Входные данные для таблицы (3):

Цвет 1: RGB в формате 5.6.5, 16 бит

Цвет 2: RGB в формате 5.6.5, 16 бит

Коэффициент альфа: 0..255, 8 бит

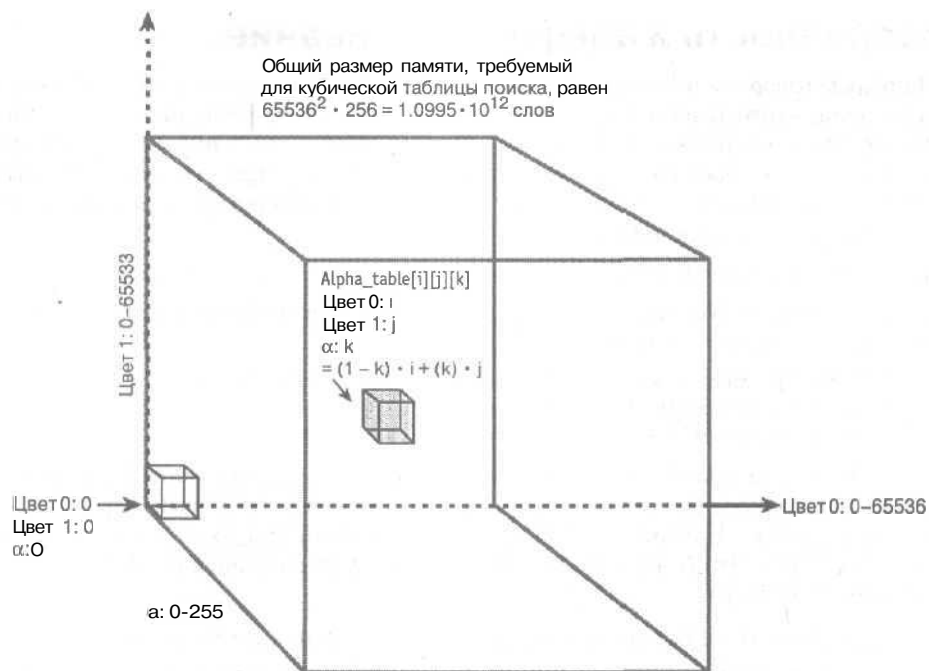


Рис. 12.4. Альфа-смешивание, реализованное с помощью таблицы

Нам нужно, чтобы таблица позволяла найти все возможные произведения двух входных цветов 1 и 2 при 256 различных коэффициентах альфа-смешивания. Следовательно, мы получаем следующий размер таблицы:  
 $65536 \cdot 65536 \cdot 256 = 1.0995 \cdot 10^{12}$

Вряд ли нужно говорить, что в таком виде этот способ работать не будет. Давайте подумаем, нельзя ли эту проблему решить иначе. Рассмотрим математическую часть еще раз:

```
p_finalr = (alpha)*(p_source1r) + (1-alpha)*(p_source2r)
p_finalg = (alpha)*(p_source1g) + (1-alpha)*(p_source2g)
p_finalb = (alpha)*(p_source1b) + (1-alpha)*(p_source2b)
```

Если мы откажемся от решения всей проблемы сразу, то сможем сконцентрироваться на некоторых частных проблемах, которые стоят того. Основные вычисления — это умножение коэффициента  $\alpha$  на каждый из RGB-компонентов, а также умножение значения  $(1-\alpha)$  на те же компоненты. Давайте создадим таблицу поиска из 65536 строк, в каждой из которых записывается произведение коэффициента  $\alpha$  (например,  $\alpha=0.5$ ) на значения RGB-компонент, записанных в формате 5.6.5. Рис. 12.5 иллюстрирует этот подход.

Если у нас есть такая таблица, то чтобы узнать, как смешиваются два пикселя с коэффициентом смешивания  $\alpha=0.5$ , нужно использовать формулу:

```
p_final = (0.5)*(p_source1) + (1-0.5)*(p_source2)
```

Провести вычисления по ней можно с использованием двух таблиц поиска и одной операции сложения:

```
p_final = alphatable[p_source1] + alphatable[p_source2];
```

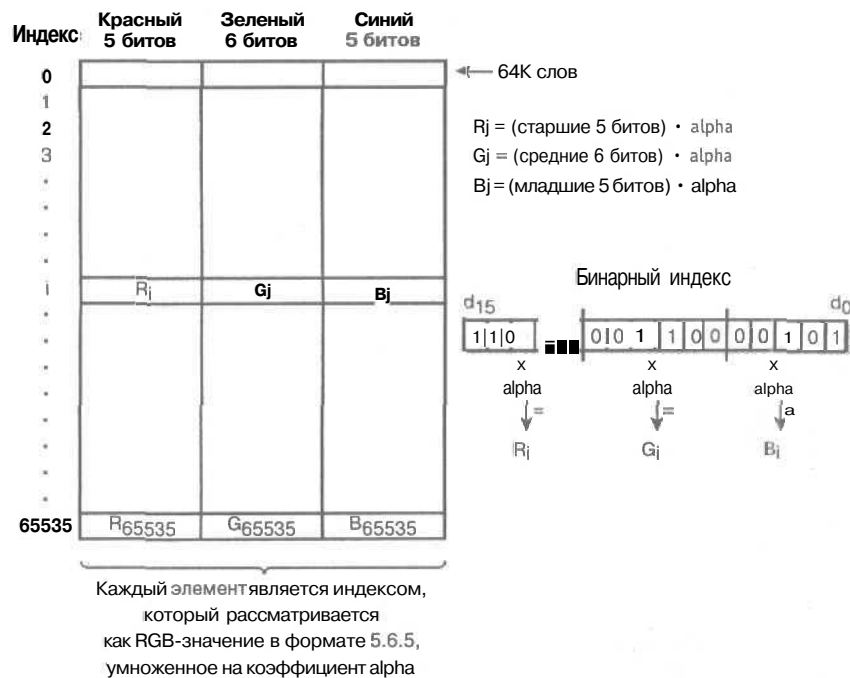


Рис. 12.5. Более эффективная таблица поиска

Итак, имея одну таблицу поиска, содержащую 65 536 значений, каждое из которых состоит из 16 бит, мы можем заменить все формулы типа

$$\begin{aligned} p\_finalr &= (\alpha) \cdot (p\_source1r) + (1-\alpha) \cdot (p\_source2r) \\ p\_finalg &= (\alpha) \cdot (p\_source1g) + (1-\alpha) \cdot (p\_source2g) \\ p\_finalb &= (\alpha) \cdot (p\_source1b) + (1-\alpha) \cdot (p\_source2b) \end{aligned}$$

такой одной формулой:

$$p\_final = alphatable[p\_source1] + alphatable[p\_source2];$$

Скажу вам сразу, этот метод работает — и очень быстро! Вы можете спросить о других значениях коэффициента alpha. Ответу: каждому значению коэффициента alpha соответствует таблица размером 64Kb, поэтому если мы хотим ввести поддержку восьми равноотстоящих значений коэффициента alpha — 0, 0.125, 0.250, 0.375, 0.5, 0.625, 0.75, 0.875, 1.00, — то нам нужно девять таблиц общим размером  $9 \cdot 64 \text{ Кбайт} = 576 \text{ Кбайт}$ , а это уже почти нормально.

#### НА ЗАМЕТКУ

Надеюсь, вы сразу поймете, что нам не нужны таблицы со значениями коэффициента 0 и 1.0. Для этих случаев мы можем использовать условную логику. Если  $\alpha = 0$ , пиксель полностью прозрачен, и мы можем просто использовать другой пиксель без каких-либо сложностей. Аналогично, если  $\alpha = 1$ , мы выводим пиксель как обычно, без какого-либо беспокойства об alpha, поскольку при этом пиксель полностью непрозрачен. Сейчас мы жертвуем дополнительной памятью, чтобы не загружать альфа-растеризатор обработкой особых случаев.

В заключение давайте создадим таблицу поиска для набора значений коэффициента alpha в диапазоне 0—1, которая позволит нам вычислять произведения значений alpha на все возможные RGB-значения. При этом значения r, g, b будем считать представленными в формате 5.6.5. Мы хотим умножать alpha не на целое 16-битовое число из диапазона

0—65535, а на отдельные извлеченные значения  $r$ ,  $g$ ,  $b$ , представленные целым числом. В псевдокоде этот алгоритм выглядит следующим образом.

for каждый цвет в диапазоне 0-65535

**begin**

Преобразуем каждое 16-битовое значение  
в  $r,g,b$ -формат 5.6.5

Теперь умножим значение каждого канала на  $\alpha$

$r=r*\alpha$   
 $g=g*\alpha$   
 $b=b*\alpha$

Помещаем 16-битовое слово **цвета**, построенного из  
 $r,g,b$ -значений, в **таблицу** поиска, индексированную  
цветом:

$\text{alphatable}[\text{value}] = r.g.b$

**end**

Создать такую таблицу довольно просто. Ключевым моментом является то, что мы используем 16-битовый индекс для представления цвета, который мы хотим умножить на коэффициент  $\alpha$ . Однако перед тем как выполнить умножение, мы должны извлечь  $rgb$ -значения из каждого 16-битного значения, **имеющего** вид обычного числа, умножить их на значение  $\alpha$ , а затем снова составить их вместе. После этого нам понадобится несколько таких таблиц, возможно, 8 или 16, чтобы можно было работать с достаточным количеством коэффициентов  $\alpha$ . Я выбрал значение 8, но вы можете изменить его, Таблица поиска хранится в `T3DLIB10.CPP` следующим образом.

```
USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536];
```

Сейчас `NUM_ALPHA_LEVELS` присвоено значение 8, что дает 8 уровней  $\alpha$  с шагом 0.125. Таблица располагается построчно, **каждая** строка соответствует отдельному значению  $\alpha$  (см. рис. 12.6).

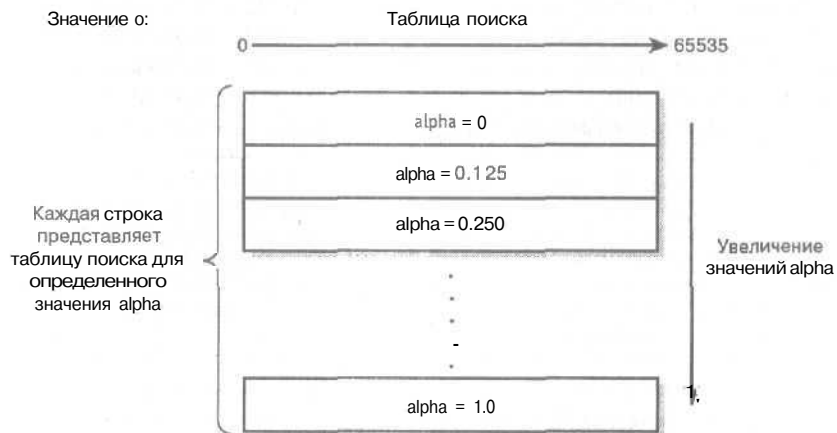


Рис. 12.6. Использование нескольких таблиц поиска

Ниже приводится код функции `RGB_Alpha_Table_Builder()`, которая строит все эти таблицы.

```
int RGB_Alpha_Table_Builder(
    int num_alpha_levels, // Количество уровней alpha
    // Таблица поиска
    USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536])
{
    // Эта функция создает таблицу поиска для выполнения
    // альфа-смешивания. При альфа-смешивании двух
    // RGB-значений можно просто использовать формулу
    // final = (alpha)*(source1RGB) + (1-alpha)*(source2RGB)

    // Работа функции заключается в создании таблицы,
    // состоящей из num_alpha_levels строк, каждая из
    // которых состоит из 65536 значений, представляющих
    // собой произведение масштабного коэффициента на
    // значение каждого из 65536 цветов в RGB-формате

    // Значение num_alpha_levels должно быть степенью
    // двойки. Например, num_alpha_levels - 8 даст таблицу
    // множителями которых будут числа 8/8, 7/8, 6/8 ...
    // 1/8, 0/8, представляющие собой коэффициенты
    // прозрачности пикселей в смешивании

    // Вначале проверяем указатель
    if (!rgb_alpha_table)
        return(-1);

    int r,g,b; // Используется, чтобы извлекать цвета из
    // rgbindex
    float alpha = 0;
    float delta_alpha = EPSILON_E6 +
        1/((float)(num_alpha_levels-1));

    // Нам надо num_alpha_level строк
    for (int alpha_level = 0; alpha_level < num_alpha_levels;
        alpha_level++)
    {
        // Есть 65536 RGB-значений, которые нужно обработать
        for (int rgbindex = 0; rgbindex < 65536; rgbindex++)
        {
            // Извлекаем r,g,b из rgbindex в формате 5.6.5
            _RGB565FROM16BIT(rgbindex, &r, &g, &b);

            // Масштабируем
            r = (int)((float)r * (float)alpha);
            g = (int)((float)g * (float)alpha);
            b = (int)((float)b * (float)alpha);

            // Строим значение и сохраняем его
            rgb_alpha_table[alpha_level][rgbindex] =
                _RGB16BIT565(r,g,b);
        } // rgbindex
    }
```

```

    // Увеличиваем альфа
    alpha+=delta_alpha;
} //forrow

// Возврат кода успешного завершения
return (1);

} // RGB_Alpha_Table_Builder

```

Для использования функции мы вызываем ее с необходимым количеством альфа-уровней, и создаем таблицу (которая в нашем случае является глобальной с фиксированным размером). Вызов функции выглядит следующим образом.

```
RGB_Alpha_Table_Builder(NUM_ALPHA_LEVELS, rgb_alpha_table);
```

Теперь, когда у нас есть таблица поиска, давайте посмотрим, как ею воспользоваться.

## Добавление альфа-смешивания в базовые растеризаторы

Добавление альфа-смешивания в базовые растеризаторы без Z-буферизации — это путь наименьшего сопротивления, так что начнем с него. Вначале посмотрим, где мы хотим использовать альфа-смешивание. Основная цель альфа-смешивания — дать возможность увидеть один объект визуализации сквозь другой. Эта концепция проиллюстрирована на рис. 12.7.

У нас есть сцена. Производится визуализация объекта этой сцены с коэффициентом альфа, равным 20%. Следовательно, фон составляет 80% результирующего цвета, а объект — 20%. Это приводит нас к первой проблеме альфа-смешивания и прозрачности: объект под прозрачным объектом должен быть нарисован первым. Это общая проблема для ряда алгоритмов. Допустим, у нас есть оптимальный алгоритм визуализации с нулевой перерисовкой пикселей. Это означает, что любой многоугольник, закрытый другим многоугольником, не рисуется (рис. 12.8). Проблема состоит в том, что если мы захотим, чтобы многоугольник, находящийся поверх других объектов, стал прозрачным, под ним ничего не будет видно!

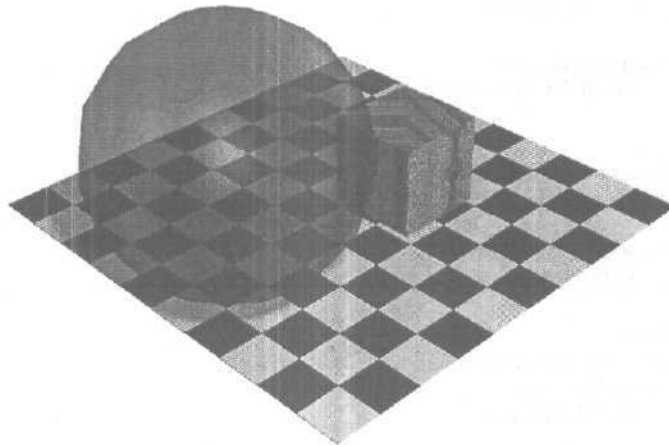


Рис. 12.7. Альфа-смешивание в действии

Другая проблема связана с Z-буферизацией. Предположим, два многоугольника попадают в Z-буфер в последовательности передний-задний, т.е. многоугольник 1 ближе к наблюдателю, чем многоугольник 2 (рис. 12.9),

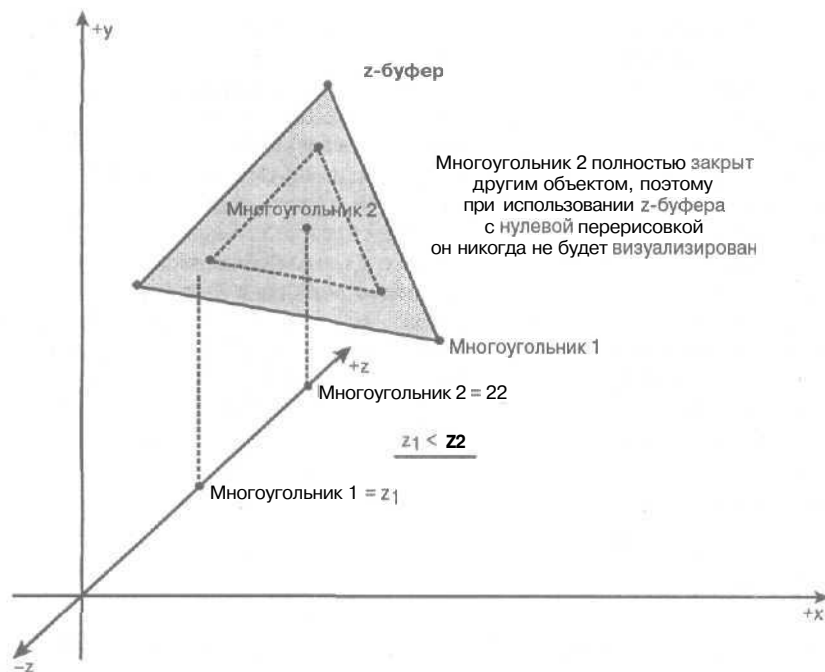


Рис. 12.8. Многоугольники, закрытые другими объектами, не визуализируются

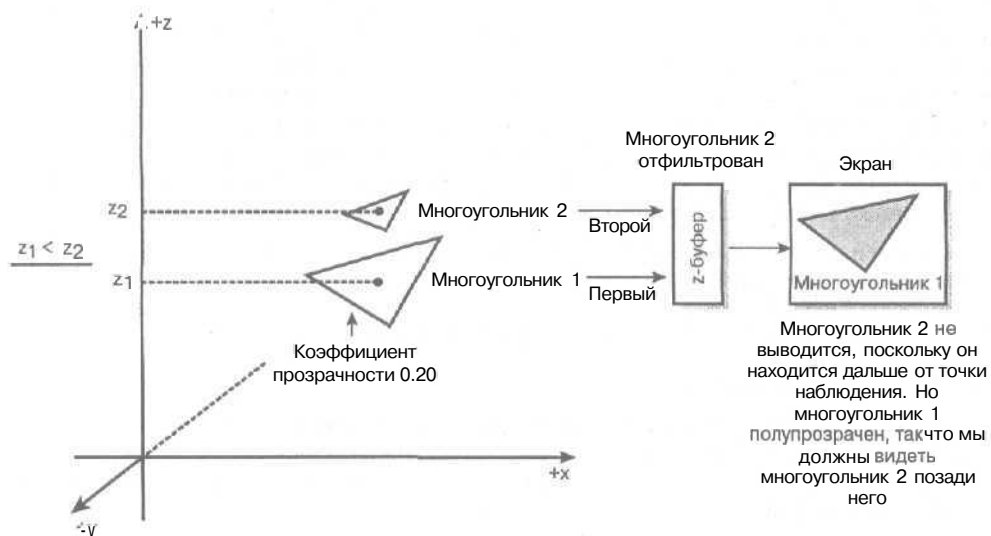


Рис. 12.9. Проблемы, связанные с альфа-смешиванием и Z-буферизацией

В этом случае многоугольник 1 проходит тест Z-буферизации и записывается на экран и в Z-буфер с прозрачностью. Однако многоугольник 2 не проходит тест Z-буферизации и не записывается вообще. Но многоугольник 1 полупрозрачный, поэтому многоугольник 2 должен быть видим сквозь него. Как видите, у нас проблема. Есть несколько способов решения таких проблем. Например, если вы стремитесь достичь ну-

левой степени перерисовки, прозрачные многоугольники не должны принимать участия в вычислениях, т.е. они выводятся в любом случае. Что касается проблемы Z-буфера, то мы можем сортировать многоугольники от задних к передним, чтобы получить грубую упорядоченность, а затем передать их в 2-буфер, чтобы получить точный порядок с учетом альфа-прозрачности. При этом дальние многоугольники будут нарисованы раньше, а более близкие — позже, и тем самым будет выполнено смешивание. Если вы не хотите терять время на сортировку многоугольников, то можете пометить прозрачные многоугольники, а затем, при визуализации, если выясняется, что какой-либо многоугольник полностью находится в границах более близкого к точке наблюдения прозрачного многоугольника, то он выводится первым **независимо от** порядка в Z-буфере.

Итак, начнем с простейшего растеризатора, **выводящего** многоугольники с постоянным/плоским затенением без использования Z-буфера или текстур. Это новая функция `Draw_Triangle_2D3_16()`. Мы можем изменить ее внутренний цикл и добавить код для поддержки альфа-прозрачности в качестве параметра. Давайте **еще** раз посмотрим на ее внутренний цикл и составим план действий.

```
// Выводим строку
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель в формате 5.6.5
    screen_ptr[xi] = color;
} // for xi
```

Чтобы ввести прозрачность, мы должны прочесть текущий пиксель `screen_ptr[xi]` и смешать его с определенным цветом, вместо того, чтобы записывать этот цвет в `screen_ptr[xi]`. Для этого мы будем использовать нашу таблицу поиска. **Предположим**, что таблица всегда является глобальной, т.е. нам не нужно передавать ее функции в качестве параметра. Далее предположим, что для **общности** мы будем передавать уровень альфа в функцию растеризации. Значения параметра находятся в диапазоне от 0 до `(NUM_ALPHA_LEVELS-1)`. Затем мы будем использовать уровень альфа как индекс строки таблицы поиска и получать соответствующие указатели для строк `alpha` (напомню, что каждая строка представляет таблицу для отдельного значения альфа) и `(1-alpha)`. Если взять уровень альфа, равный 0, нам потребуется указатель на строку 0 и на строку `(NUM_ALPHA_LEVELS-1)`. Код для обращения к корректной строке таблицы для каждого источника выглядит следующим образом.

```
// Присваиваем значения указателей на основе значения
// параметра альфа многоугольника
USHORT *alpha_table_src1 -
    (USHORT*)&rgb_alpha_table[(NUM_ALPHA_LEVELS-1)-alpha][0];
USHORT *alpha_table_src2 -
    (USHORT*)&rgb_alpha_table[alpha][0];
```

Мы уже сделали большую часть работы. **Все**, что нам нужно, это выполнить смешивание пикселя на экране с пикселем, который мы собираемся вывести на экран. Для этого нужно использовать алгоритм смешивания с таблицей поиска. Ниже приведен код нового внутреннего цикла.

```
for (xi=xstart; xi < xend; xi++)
{
    // Записываем текстель в формате 5.6.5
    screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
        alpha_table_src2[color];
} // for xi
```

Мы свели сложную проблему альфа-смешивания в реальном времени с 16-битным полным цветом к одной операции сложения и двум поискам в таблице. Здорово, правда?

Давайте теперь посмотрим, как добавить альфа-смешивание к каждому одиночному растеризатору. Следует определить таблицы с альфа-уровнем, прочитать пиксель и смешать его с пикселем или текстелем, который готовится к выводу на экран, — и это все!

## Поддержка альфа-смешивания для простых растеризаторов

Конечная функция с поддержкой альфа-смешивания, которая выполняет визуализацию многоугольников без Z-буферизации с постоянным/плоским затенением называется `Draw_Triangle_2D_Alpha16()`. Вот ее прототип.

```
void Draw_Triangle_2D_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    int alpha);           // Используемый уровень альфа
```

За исключением добавленного параметра альфа, функция работает точно так же, как и ее версия без поддержки альфа-смешивания, с теми же параметрами. Однако не забывайте, что значение альфа должно быть в диапазоне `0..(NUM_ALPHA_LEVELS-1)`, иначе аозникнет ошибка доступа к памяти, связанная с выходом за пределы диапазона таблицы поиска. Теперь давайте добавим альфа-поддержку в другие базовые растеризаторы, поддерживающие текстурирование и затенение по Гуро.

## Поддержка альфа-смешивания для растеризаторов с затенением по Гуро

Добавление поддержки альфа-смешивания к затенению по Гуро тривиально. Нам необходимо лишь указать таблицы поиска.

```
// Определяем таблицы для источников sourtel и
// source2 на основе уровня альфа многоугольника
USHORT *alpha_table_src1 -
    (USHORT *)&rgb_alpha_table[(NUM_ALPHA_LEVELS-1)
    - alpha][0];
USHORT *alpha_table_src2 -
    (USHORT *)&rgb_alpha_table[alpha][0];
```

Затем мы изменяем внутренний цикл, заменяя фрагмент

```
// Вывод строки
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель в формате 5.6.5
    screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
        ((vi >> (FIXP16_SHIFT+2)) << 5) +
        (wi >> (FIXP16_SHIFT+3));
```

// Интерполируем u,v,w

ui+=du;

vi+=dv;

wi+=dw;

} // for xi

следующим фрагментом

```
// Вывод строки
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель в формате 5.6.5
```

```

screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
    alpha_table_src2[((ui >> (FIXP16_SHIFT+3)) << 11) +
        ((vi >> (FIXP16_SHIFT+2)) << 5) +
        fyri >>(FIXP16_SHIFT+3))];

// Интерполируем u,v,w
ui+=du;
vi+=dv;
wi+=dw;
} // for xi

```

Обратите внимание, что теперь **текстель** из первоначального кода является параметром таблицы поиска альфа-смешивания (как и пиксель на экране). Процесс преобразования каждой такой функции почти тривиален. Для этого мы просто берем код генерации значения пикселя и используем его в качестве параметра в таблице поиска. Затем берется адрес **пикселя**, куда должно записываться результирующее значение, и используем его как адрес назначения, а также как второй параметр в таблице поиска. Вот и все, что нужно сделать. Вот прототип новой функции затенения по **Гуро** с поддержкой альфа-смешивания.

```

void Draw_Gouraud_Triangle_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch, // Байт в строке
    int alpha); // Уровень альфа

```

Единственное отличие от версии без поддержки альфа-смешивания состоит в параметре альфа, который должен лежать в диапазоне 0..(NUM\_ALPHA\_LEVELS-1).

## Поддержка альфа-смешивания для растеризаторов аффинных текстур

Добавление поддержки альфа-смешивания в растеризаторы аффинных текстур идентично предыдущим примерам, но тут мы должны создать три различных версии. Я просто перечислю прототипы.

```

void Draw_Textured_Triangle_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch, // Байт в строке, 320, 640 и т.д.
    int alpha); // Уровень альфа

```

```

void Draw_Textured_TriangleFS_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch, // Байт в строке, 320, 640 и т.д.
    int alpha); // Уровень альфа

```

```

void Draw_Textured_TriangleGS_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch, // Байт в строке, 320, 640 и т.д.
    int alpha); // Уровень альфа

```

Теперь у нас есть текстурные растеризаторы с постоянным затенением, плоским затенением, а также затенением по Гуро, с поддержкой альфа-канала (я выделил новые па-

раметры). Давайте в качестве примера рассмотрим внутренний цикл простого растеризатора с постоянным затенением до и после введения поддержки альфа-смешивания. Ниже приводится внутренний цикл растеризации объекта перед добавлением поддержки альфа-смешивания.

```
// Выводим строку
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель
    screen_ptr[xi]=textmap[(ui >> FIXP16_SHIFT)+
        ((vi >> FIXP16_SHIFT) << texture_shift2)];

    // Интерполируем u,v
    ui+=du;
    vi+=dv;
} // for xi
```

Ниже он представлен после добавления поддержки альфа-смешивания с использованием таблиц поиска.

```
// Выводим строку
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель
    screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
        alpha_table_src2[textmap[(ui >> FIXP16_SHIFT) +
            ((vi >> FIXP16_SHIFT) << texture_shift2)]];

    // Интерполируем u,v
    ui+=du;
    vi+=dv;
} // for xi
```

Опять мы просто используем текстель и экранный пиксель как параметры в таблицах поиска, а затем суммируем результаты и выводим их на экран. Наконец вот функция визуализации для вызова растеризатора с поддержкой альфа-смешивания.

```
void Draw_RENDERLIST4DV2_Solid_Alpha16(
    RENDERLIST4DV2_PTR rend_list, // Список визуализации
    UCHAR *video_buffer,          // Указатель на видеобуфер
    int lpitch,                   // Байт в строке видеобуфера
    int alpha_override);          // Переопределенное значение
```

Функция работает так же, как и все остальные. Вы передаете ей список визуализации, указатель на видеобуфер и шаг памяти, и она выполняет визуализацию согласно списку, т.е. рисует все многоугольники из структуры данных. Однако есть один интересный дополнительный параметр: `alpha_override`. Он предназначен для того, чтобы выполнять альфа-перерисовку всех объектов. Если `alpha_override` равен -1, ничего не происходит. Если `alpha_override` лежит в диапазоне `0..(NUM_ALPHA_LEVELS-1)`, то это значение будет подменять все существующие значения `alpha`. Визуализация будет выполняться с новым значением параметра, причем все остальные параметры остаются неизменными.

#### ВНИМАНИЕ

Я кое-что здесь пропустил: функции растеризации получают в качестве параметра уровень альфа, но где они его берут? В данный момент они нигде его не берут, но вскоре мы будем кодировать его в базовом цвете многоугольника.

## Добавление альфа-смешивания в растеризаторы с Z-буферизацией

Добавление альфа-смешивания в растеризаторы с Z-буферизацией выполняется точно так же, как и в случае растеризаторов без Z-буферизации. Здесь нет каких-либо отличий, за исключением отличия в логике. Многоугольники, поступающие в Z-буфер после многоугольника, уже находящегося в буфере, не будут показываться под ним, т.е. визуализация будет выполняться неправильно. Для решения этой проблемы можно предварительно отсортировать многоугольники в порядке от задних к передним с помощью **Z-сортировки** и Z-буфера, а можно ввести дополнительную логику в вызывающую функцию, выполняющую визуализацию многоугольников, чтобы вначале всегда визуализировались **многоугольники**, находящиеся в Z-буфере под прозрачными многоугольниками. Однако мы займемся этой задачей позже, а сейчас код альфа-смешивания с Z-буферизацией просто смешивает то, что **пришло** в буфер первым.

Добавление альфа-смешивания идентично добавлению в остальные функции, поэтому я просто покажу прототипы функций и один пример функции до и после внесения изменений.

```
void Draw_Textured_TriangleZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR* _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байт в строке Z-буфера
    int alpha);         // Уровень альфа
```

```
void Draw_Textured_TriangleFSZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR* _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байт в строке Z-буфера
    int alpha);         // Уровень альфа
```

```
void Draw_Textured_TriangleGSZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR * _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байт в строке Z-буфера
    int alpha);         // Уровень альфа
```

```
void Draw_Triangle_2DZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR * _dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байт в строке Z-буфера
    int alpha);         // Уровень альфа
```

```
void Draw_Gouraud_TriangleZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
```

```

    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch,         // Байт в строке Z-буфера
    int alpha);         // Уровень альфа

```

Эти функции включают все варианты затенения и освещенности, которыми мы располагаем! В качестве примера давайте рассмотрим наиболее сложный внутренний цикл до и после. Это будет растеризатор текстур с затенением по Гуро `Draw_Textured_TriangleGSZB_Alpha16()`. Исходный внутренний цикл перед добавлением поддержки альфа-смешивания берется из функции `Draw_Textured_TriangleGSZB_16()`.

```

// Выводим строку
for (xi=xstart; xi < xend; xi++)
{
    // Пишем текстель в формате 5.6.5
    // Проверяем, не находится ли текущий пиксель ближе к
    // наблюдателю по координате z, чем находящийся в буфере
    if (zi < z_ptr[xi])
    {
        // Вначале получаем текстель
        textel = textmap[(si >> FIXP16_SHIFT) +
            ((ti >> FIXP16_SHIFT) << texture_shift2)];

        // Извлекаем rgb-компоненты
        r_textel = (textel >> 11);
        g_textel = ((textel >> 5) & 0x3f);
        b_textel = (textel & 0x1f);

        // Модулируем текстель с затенением по Гуро
        r_textel *= ui;
        g_textel *= vi;
        b_textel *= wi;

        // Выполняем окончательную запись пикселя
        // Отметим, что мы выполнили операции
        // умножения r*32, g*64, b*32
        screen_ptr[xi] = ((b_textel >> (FIXP16_SHIFT+8)) +
            ((g_textel >> (FIXP16_SHIFT+8)) << 5) +
            ((r_textel >> (FIXP16_SHIFT+8)) << 11));

        // Обновляем Z-буфер
        z_ptr[xi] = zi;
    } // if

    // Интерполируем u,v,w,z,s,t
    ui += du;
    vi += dv;
    wi += dw;

    zi += dz;
    si += ds;
    ti += dt;
} // for xi

```

Ниже представлен **КОД ДЛЯ** добавления альфа-смешивания.

```
// Выводим строку
for (xi=xstart; xi < xend; xi++)
f
    // Пишем текстель в формате 5.6.5
    // Проверяем, не находится ли текущий пиксель ближе к
    // наблюдателю по координате z, чем находящийся в буфере
    if (zi < z_ptr[xi])
    {
        // Вначале получаем текстель
        textel = textmap[(si >> FIXP16_SHIFT) +
            ((ti >> FIXP16_SHIFT) << texture_shift2)];

        // Извлекаем rgb-компоненты
        r_textel = ((textel >> 11) );
        g_textel = ((textel >> 5) & 0x30;
        b_textel = (textel & 0x1f);

        // Модулируем текстель с затенением по Гуро
        r_textel*=ui;
        g_textel*=vi;
        b_textel*=wi;

        // Выполняем окончательную запись пикселя
        // Отметим, что мы выполнили операции
        //умножения r*32, g*64, b*32
        screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
            alpha_table_src2[((b_textel>>(FIXP16_SHIFT+8))+
                ((g_textel >>(FIXP16_SHIFT+8)) << 5) +
                ((r_textel >>(FIXP16_SHIFT+8)) << 11))];

        // Обновляем Z-буфер
        z_ptr[xi] = zi;
    } // if

    //Интерполируем u,v,w,z,s,t
    ui+=du;
    vi+=dv;
    wi+=dw;

    zi+=dz;
    si+=ds;
    ti+=dt;
} // for xi
```

**НА ЗАМЕТКУ**

Прошу прощения за код, но его очень много — более 50000 строк, и я показал только малую его часть.

Наконец, есть функция визуализации, которая вызывает все **растеризаторы** с поддержкой альфа-смешивания и **Z-буферизацией**. Она называется **Draw\_RENDERLIST4DV2\_SolidZB\_Alpha16()**. Ниже представлен ее прототип.

```
void Draw_RENDERLIST4DV2_SolidZB_Alpha16(
    RENDERLIST4DV2_PTR rendjist
    UCHAR *video_buffer,
```

```
int lpitch,
UCHAR *zbuffer,
int zpitch,
int alpha_override);
```

Функция получает обычный набор параметров — список визуализации, видеобuffer, шаг памяти, Z-буфер, шаг памяти Z-буфера. Но есть и добавленный параметр `alpha_override`, приводящий к добавлению альфа-смешивания для списка визуализации. Он может принимать значения в диапазоне `0..(NUM_ALPHA_LEVELS-1)`. В случае значения `-1` поддержка альфа-смешивания не выполняется.

Для того чтобы посмотреть на применение альфа-смешивания, запустите демонстрационную программу `DEMO112_2.CPP|EXE`. Копия экрана данной программы показана на рис. 12.10.

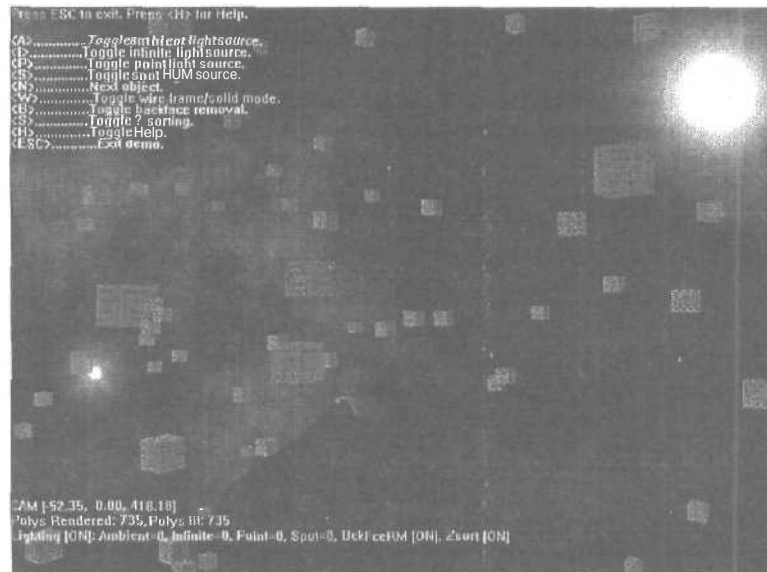


Рис. 12.10. Копия экрана демонстрационной программы с поддержкой альфа-смешивания

Эта демонстрационная программа позволяет вам двигаться в пространстве, заполненном простыми объектами (предполагается, что это будут вражеские корабли). Программа медленно увеличивает и уменьшает альфа-смешивание с помощью переменной `alpha_override` на низком уровне (поскольку мы все еще не можем устанавливать параметр альфа на уровне объектов). Вы можете также включить/отключить Z-сортировку, а также другие опции. Для компиляции демонстрационной программы убедитесь, что вы добавили в проект все модули `T3DLIB1-10.CPP|H`, а также библиотечные файлы `DirectX`.

## Поддержка альфа-смешивания на уровне объектов

Следующая проблема, с которой нам предстоит разобраться, — это способ определения многоугольников в наших моделях, который обеспечивает поддержку альфа-смешивания. По иронии судьбы мы задумались над этим только после того, как создали дескриптор цвета в `T3DLIB6.CPP|H`.

```
typedef struct RGBAV1_TYP
{
    union
    {
        int rgba; // Сжатый формат
        UCHAR rgba_M[4]; // Формат массива
        struct
        {
            UCHAR a,b,g,r; // Формат с явным указанием имени
        };
    }; // union
} RGBAV1, *RGBAV1_PTR;
```

Структура представляет полный 32-битовый цвет с поддержкой 8-битового альфа-канала и формата RGB 8.8.8. К сожалению, структуры многоугольников не используют этот тип — они используют 32-битовые целые числа для хранения цветов в сокращенном 16-битовом формате без какой-либо поддержки концепции альфа-смешивания. Тем не менее, наши дела не так уж плохи. Все RGB-цвета хранятся в 16-битовых значениях в младших 16 битах поля цвета в структурах многоугольников. Таким образом, верхние 16 битов мы можем использовать по своему усмотрению. Поэтому альфа-канал мы будем хранить в старших 8 битах адресного пространства цвета каждого многоугольника. Кроме того, мы будем помечать каждый многоугольник, имеющий уровень альфа, следующим атрибутом:

```
#define POLY4DV2_ATTR_TRANSPARENT 0x0002
```

Этот атрибут уже определен в `T3DLIB7.CPP`, поскольку мы уже заранее запланировали эту поддержку. Все, что нам нужно для поддержки альфа-смешивания на уровне многоугольников, — это просто установить для многоугольника атрибут `POLY4DV2_ATTR_TRANSPARENT` наряду с другими атрибутами, а затем используем старшие 8 битов цвета для альфа-уровня ( $0..(\text{NUM\_ALPHA\_LEVELS}-1)$ ), как показано на рис. 12.11.

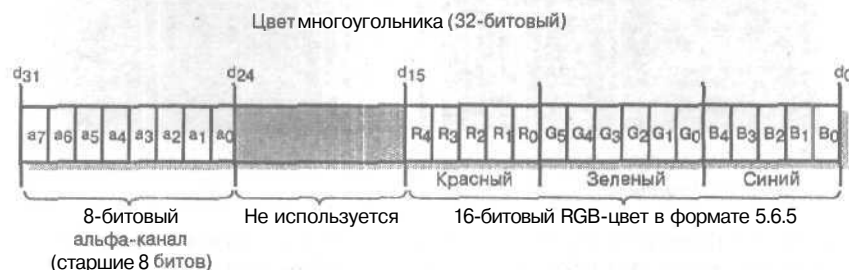


Рис. 12.11. Данные уровня альфа хранятся в старших 8 битах слова цвета

Теперь о трудностях. В нашем случае трудность заключается в том, что нужно переписать загрузчик объекта, и нет никаких способов избежать этого. Мы должны не просто переписать его, но и предложить способ маркировки многоугольника как прозрачного. К счастью для нас, `.COB`-формат программы `Caligari trueSpace` содержит класс прозрачности и мы можем им воспользоваться. Как всегда, нам нужно прийти к соглашению об использовании поля класса для передачи нужной информации, причем передачи "полуестественным" способом. Я не хочу использовать, например, розовый цвет как признак прозрачности!

Первый шаг заключается в определении того, какая поддержка прозрачности существует в различных версиях программы трехмерного моделирования `Caligari`. Похоже, вер-

сии 4.0 и старше поддерживают именно то, что нам нужно, так что можно приступать к работе. Вспомним, как мы писали код для моделей затенения. Наши соглашения по этому поводу перечислены ниже.

1. Все многоугольники должны быть треугольниками.
2. Класс цвета многоугольника без текстуры должен быть определен как `plain color`.
3. Класс цвета многоугольника с текстурой должен быть определен как `texture map`. Кроме того, для одного объекта разрешена только одна текстура, которая должна быть обернута вокруг объекта с коэффициентами повторения 1.0, 1.0.
4. Режим затенения многоугольника кодируется в классе `reflectance`: `constant` — для постоянного или излучающего затенения, `matte` для плоского затенения, и `plastic` для затенения по Гуро.

Такой подход работает великолепно. С его помощью мы можем выполнять естественное моделирование объектов и получать все, что мы хотим. Чтобы добавить прозрачность, мы будем использовать класс прозрачности из `.COB`-формата *Caligari*. На рис. 12.12 показана копия экрана *Caligari trueSpace* в момент выбора уровня прозрачности.

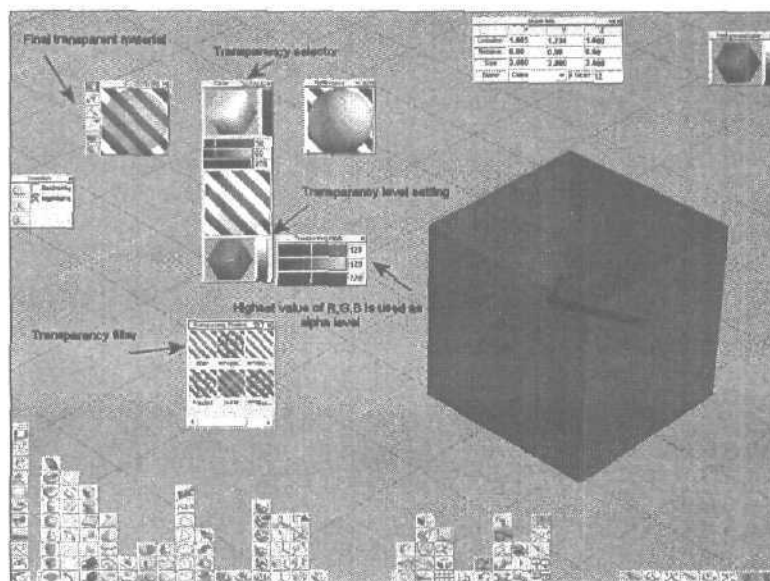


Рис. 12.12. Копия экрана *Caligari trueSpace* при установке уровня прозрачности

Мы устанавливаем прозрачность в режим "filter", открываем панель инструментов прозрачности, а затем используем ползунок для установки уровня прозрачности (величины альфа). Например, у куба на рис. 12.12 установлен класс цвета `plain color` (текстурирование отсутствует), отражающая способность `matte` (плоское затенение) и прозрачность в режиме `filter`, а также RGB-параметр 128 из 256, т.е. 50%. Ниже приводятся данные, которые система выводит в файл `CUBEBLUEALPHA_01.COB` (без некоторых дополнительных данных, не интересующих нас в настоящий момент).

```
Caligari V00.01ALH
Name Cube
center 1.08471 1.23355 1
```

```

x axis 1 0 0
y axis 0 1 0
z axis 0 0 1
Transform
1 0 0 1.08471
0 1 0 1.23355
0 0 1 0
0 0 0 1
World Vertices 8
-1.000000 -1.000000 0.000000
-1.000000 -1.000000 2.000000
1.000000 -1.000000 0.000000
1.000000 -1.000000 2.000000
-1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
1.000000 1.000000 2.000000
-1.000000 1.000000 2.000000
Texture Vertices 14
0.000000 0.333333
0.000000 0.666667
0.250000 0.333333
0.250000 0.666667
0.500000 0.000000
0.500000 0.333333
0.500000 0.666667
0.500000 1.000000
0.250000 0.000000
0.250000 1.000000
0.750000 0.333333
0.750000 0.666667
1.000000 0.333333
1.000000 0.666667
Faces 12
Face verts 3 flags 0 mat 0
<0,0> <1,1> <3,3>
Face verts 3 flags 0 mat 0
<0,0> <3,3> <2,2>
Face verts 3 flags 0 mat 0
<0,8> <2,2> <5,5>
Face verts 3 flags 0 mat 0
<0,8> <5,5> <4,4>
Face verts 3 flags 0 mat 0
<2,2> <3,3> <6,6>
Face verts 3 flags 0 mat 0
<2,2> <6,6> <5,5>
Face verts 3 flags 0 mat 0
<1,9> <7,7> <6,6>
Face verts 3 flags 0 mat 0
<1,9> <6,6> <3,3>
Face verts 3 flags 0 mat 0
<4,10> <5,5> <6,6>
Face verts 3 flags 0 mat 0
<4,10> <6,6> <7,11>
Face verts 3 flags 0 mat 0

```

```

<0,12> <4,10> <7,11>
Face verts 3 flags 0 mat 0
<0,12> <7,11> <1,13>
DrawFlags0
Unit V0.01 Id 18620117 Parent 18620116 Size 00000009
Units 2
ObRQ V0.01 Id 18620121 Parent 18620116 Size 00000121
Object RadiosityQuality: 0
Object Radiosity Max Area: 0
Object Radiosity Min Area: 0
Object Radiosity Mesh Accuracy: 0
Mat1 V0.06 Id 18855732 Parent 18620116 Size 00000099
mat# 0
shader: phong facet: auto32
rgb 0.227451,0.235294,1
alpha 1 ka 0.1 ks 0.1 exp 0 ior 1
ShBxV0.03 Id 18855733 Parent 18855732 Size 00000462
Shaderclass: color
Shader name: "plain color" (plain)
Number of parameters: 1
colour: color (58, 60, 255)
Flags: 3
Shader class: transparency
Shader name: "filter" (plain)
Number of parameters: 1
colour: color (128, 128, 128)
Flags: 3
Shader class: reflectance
Shader name: "matte" (matte)
Number of parameters: 2
ambient factor: float 0.1
diffuse factor: float 1
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
END V1.00 Id 0 Parent 0 Size 0

```

Я выделил все классы, которые мы используем. Нам необходимо добавить в анализатор .COB-файлов код для считывания еще одного класса (transparency) и разбора поля colour для извлечения уровня альфа. Но прежде чем мы это сделаем, нам нужно уточнить еще один момент. Прозрачность в большинстве программ моделирования (в том числе и trueSpace) имеет три канала: красный, зеленый и синий. У нас не будет поддержки альфа-смешивания в многоцветном режиме, мы будем использовать только монохромный альфа-канал, в качестве которого выберем наиболее интенсивный из трех цветовых каналов. Анализатор будет выбирать наибольшее значение из трех компонент и делать его альфа-значением. Поскольку альфа-значение из .COB-файла будет в диапазоне 0..255, мы должны масштабировать его в наш диапазон (0..(NUM\_ALPHA\_LEVELS-1)). Ниже приводится прототип нового .COB-загрузчика с поддержкой альфа-смешивания.

```

int Load_OBJECT4DV2_COB2(
    OBJECT4DV2_PTR obj, // Указатель на объект
    char *filename,      // Имя .COB-файла

```

```

VECTOR4D_PTR scale,      // Коэффициенты начального
                          // масштабирования
VECTOR4D_PTR pos,        // Начальное положение
VECTOR4D_PTR rot          // Начальный поворот
int vertex_flags,         // Флаги переупорядочения вершин
                          // и выполняемых преобразований
int mipmap )              // Флаг разрешения множественного
                          // отображения (0 - его
                          // отсутствие, 1 - наличие)

```

Сейчас мы игнорируем параметр множественного **отображения**, который обсудим позднее, и просто посмотрим на прототип. Заметьте, что за исключением параметра **множественного отображения**, прототип идентичен предыдущей версии `Load_OBJECT4DV2_COB()`. Изменения касаются только чтения класса прозрачности **.COB-модели**. Давайте посмотрим на код.

```

////////////////////////////////////
// Добавленный код для прозрачности и альфа-смешивания //
////////////////////////////////////
// Сейчас нам нужно знать, есть ли прозрачность материала,
// что кодируется в классе transparency. Кроме того, мы
// должны использовать режим filter и установить RGB-цвет,
// соответствующий требуемому уровню прозрачности —
// 0,0,0 означает полную прозрачность,
// 255,255,255 — полную непрозрачность

// Поиск строки "Shader class: transparency"
while(1)
{
    // Получаем следующую строку
    if (!parser.Getline(PARSER_STRIP_EMPTY_LINES |
        PARSER_STRIP_WS_ENDS))
    {
        Write_Error("\nshader transparency class not found"
            " in .COB file %s.", filename);
        return(0);
    } // if

    // Строка "Shader class: transparency"?
    if (parser.Pattern_Match(parser.buffer,
        "[Shader] [class:] [transparency]"))
    {
        // Теперь мы знаем, что дальше следует строка
        // "Shader name:" - это именно то, что мы ищем, так
        // что мы осуществляем выход из цикла
        break;
    } // if
} // while

while(1)
{
    // Получаем следующую строку
    if (!parser.Getline(PARSER_STRIP_EMPTY_LINES |
        PARSER_STRIP_WS_ENDS))
    {

```

```

Write_Error("\nshader name ended abruptly!"
            "in .COB file %s.", filename);
return (0);
} // if

// Освобождаемся от кавычек
ReplaceChars(parser.buffer, parser.buffer, "\"", "", 1);

// Нашли ли мы имя?
if (parser.Pattern_Match(parser.buffer,
                        "[Shader] ['name:'] [s>0]"))
{
    // Определяем, разрешена ли прозрачность
    if (strcmp(parser.pstrings[2], "none") == 0)
    {
        // Сбрасываем альфа-бит, присваиваем альфа
        // значение 0
        RESET_BIT(materials[material_index +
                        num_materials].attr,
                  MATV1_ATTR_TRANSPARENT);

        // Устанавливаем alpha равным 0
        materials[material_index + num_materials].
            color.a = 0;
    } // if

    else if (strcmp(parser.pstrings[2], "filter") == 0)
    {
        // Устанавливаем альфа-бит и записываем уровень
        // альфа
        SET_BIT(materials[material_index +
                        num_materials].attr,
               MATV1_ATTR_TRANSPARENT);

        // Теперь ведем поиск строки цвета, чтобы
        // извлечь уровень альфа
        // Ищем "Shader class: transparency"
        while(1)
        {
            // Получаем следующую строку
            if (!parser.Getline(PARSER_STRIP_EMPTY_LINES|
                               PARSER_STRIP_WS_ENDS))
            {
                Write_Error("\ntransparency color not"
                            " found in .COB file %s.",
                            filename);
                return (0);
            } // if

            // Освобождаемся от посторонних символов
            ReplaceChars(parser.buffer, parser.buffer,
                        ":(,)", "", 1);

            // Поиск цвета
            if (parser.Pattern_Match(parser.buffer,
                                    "[colour] ['color'] [0] [1] [i]"))

```

```

!
// Устанавливаем для альфа-уровня
// наивысшее значение
int max_alpha = MAX(parser.pints[0],
    parser.pints[1]);
max_alpha = MAX(max_alpha,
    parser.pints[2]);
// Устанавливаем значение альфа
materials[material_index +
    num_materials].color.a =
    (int)((float)max_alpha/255 *
    (Float)(NUM_ALPHA_LEVELS-1) +
    (float)0.5);

if (materials[material_index +
    num_materials].color.a >=
    NUM_ALPHA_LEVELS)
    materials[material_index +
    num_materials].color.a =
    NUM_ALPHA_LEVELS-1;
break;
} // if
} // while
} // if
break;
} // if
} // while

```

Я выделил процесс окончательного преобразования альфа-уровня. Последнее, что нужно сделать, — это в конце загрузчика установить бит прозрачности многоугольника, чтобы при растеризации был вызван растеризатор с поддержкой альфа-смешивания, а не растеризатор для непрозрачных объектов. Ниже приведен соответствующий код. Обратите внимание на расположение альфа-канала в старших 8 битах.

```

// Проверка альфа-канала
if (materials[poly_material[curr_poly] ].attr &
    MATV1_ATTR_TRANSPARENT)
{
    // Устанавливаем значение альфа-канала
    // Для его хранения используются старшие 8 битов цвета.
    // Будем надеяться, что это не приведет к сбоям в работе
    // процессора освещения
    obj->plist[curr_poly].color +=
        (materials[ poly_material[curr_poly] ].color.a<<24);

    // Устанавливаем альфа-флаг многоугольника
    SET_BIT(obj->plist[curr_poly].attr,
        POLY4DV2_ATTR_TRANSPARENT);
} // if

```

Вот и все. Мы уже видели реализацию альфа-смешивания на основе списка визуализации, но у нас не было возможности управления на уровне объектов — мы просто задавали значение `alpha_override` и вызывали функцию визуализации списка. При этом вы-

полнялась замена всех параметров, и прозрачными становились все многоугольники. Сейчас мы можем ввести поддержку альфа-смешивания на уровне отдельных многоугольников. В качестве примера управления прозрачностью познакомьтесь с демонстрационной программой DEMO112\_3.CPP|EXE. В ней имеется ряд прозрачных сфер, а также непрозрачная сфера с шахматной черно-белой текстурой. Сферы двигаются и перекрывают друг друга. На рис. 12.13 показана копия экрана этой программы. Как всегда, в программе доступна экранная справка. Попробуйте убрать рассеянное освещение и бесконечно удаленный источник света с помощью соответствующих клавиш <A> и <L> и посмотреть на результат.

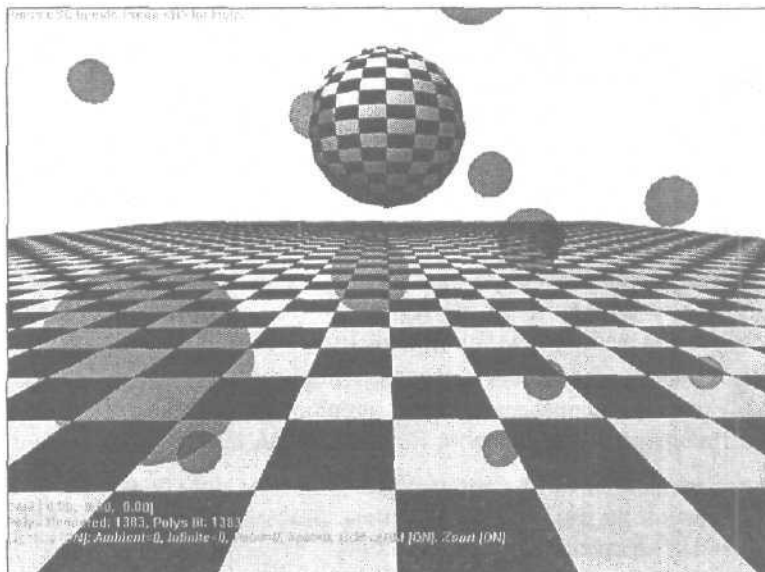


Рис. 12.13. Копия экрана демонстрационной программы альфа-смешивания на уровне объектов

## Добавление поддержки альфа-смешивания в генератор ландшафта

Теперь, когда мы можем управлять альфа-смешиванием на уровне многоугольников, давайте добавим его в генератор ландшафта. Один из лучших примеров использования прозрачности — это подводный ландшафт, который позволяет протестировать прозрачность в большом масштабе. Я разработал имитатор водного мотоцикла с поддержкой прозрачности воды. Для этого можно использовать карту цветов или карту высот и реализовать прозрачность путем придания многоугольникам атрибута прозрачности и некоторого параметра интенсивности прозрачности. Но легче просто определить параметр "уровня моря" и уровень альфа и передать их функции генерации ландшафтов, которая сгенерирует сетку ландшафта. При этом любой многоугольник ниже уровня моря делается прозрачным со значением альфа, соответствующим его цвету. На изменение генератора ландшафта и обеспечения поддержки этого эффекта уйдет пять минут. Ниже представлен новый прототип функции генератора ландшафтов версии 2.0.

```
int Generate_Terrain2_OBJECT4DV2(
    OBJECT4DV2_PTR obj, // Указатель на объект
```

```

float twidth,           // Ширина по оси x в мировых
                        // координатах
float theight,          // Высота (длина) по оси z в
                        // мировых координатах
float vscale,           // Вертикальный масштаб ландшафта
char *height_map_file, // Имя файла битового образа
                        // высот
char *texture_map_file, // Имя файла карты текстур
int rgbcolor,           // Цвет ландшафта при отсутствии
                        // текстур
VECTOR4D_PTR pos,      // Начальное положение
VECTOR4D_PTR rot        // Начальный поворот
int poly_attr,          // Атрибуты затенения
float sea_level,        // Высота уровня моря
int alpha);             // Уровень альфа, с которым
                        // визуализируются все
                        // многоугольники ниже уровня
                        // моря

```

У функции `Generate_Terrain2_OBJECT4DV2()` есть два новых параметра.

- `float sea_level` — уровень относительно высоты ландшафта, являющийся пороговым значением для прозрачности объектов. Любой многоугольник, высота которого меньше этого значения, будет прозрачным.
- `int alpha` — значение в диапазоне `0..255`, которое показывает, насколько прозрачна вода (0 — полностью прозрачна, 255 — полностью непрозрачна). Эти значения будут масштабированы до диапазона `(0..(NUM_ALPHA_LEVELS-1))`.

Для включения прозрачности устанавливаем значение `alpha` большим -1. Уровень моря `sea_level` (что бы он ни означал) будет использоваться как показатель того, нужно ли делать многоугольники прозрачными (рис. 12.14)

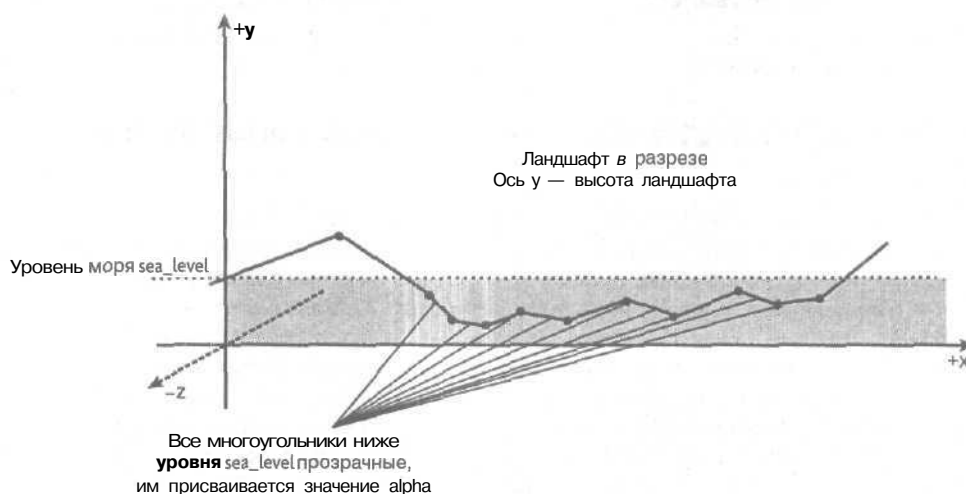


Рис. 12.14. Добавление прозрачности в генератор ландшафта

За исключением этой дополнительной функциональности, больше никаких отличий в генерации ландшафтов нет — все по-прежнему делается методом "в лоб". Ниже приво-

дится дополнительный код, поддерживающий альфа-канал. Это раздел кода генератора ландшафта, отвечающий за генерацию **вершин**.

```
// Проверяем, включена ли поддержка альфа-смешивания
// многоугольников для реализации водных эффектов
if (alpha >= 0)
{
    // Вычисляем высоты обоих многоугольников
    float avg_y_poly1 =
        (obj->vlist_local[obj->plist[poly*2].vert[0]].y +
         obj->vlist_local[obj->plist[poly*2].vert[1]].y +
         obj->vlist_local[obj->plist[poly*2].vert[2]].y)/3;
    float avg_y_poly2 =
        (obj->vlist_local[obj->plist[poly*2+1].vert[0]].y +
         obj->vlist_local[obj->plist[poly*2+1].vert[1]].y +
         obj->vlist_local[obj->plist[poly*2+1].vert[2]].y)/3;

    // Проверяем высоту многоугольника poly1 относительно
    // уровня моря
    if (avg_y_poly1 <= sea_level)
    {
        int ialpha = (int)((float)alpha/255 *
                          (float)(NUM_ALPHA_LEVELS-1) +
                          (float)0.5);
        // Устанавливаем альфа-цвет
        obj->plist[poly*2+0].color+= (ialpha < 24);
        // Устанавливаем в многоугольнике альфа-флаг
        SET_BIT(obj->plist[poly*2+0].attr,
                POLY4DV2_ATTR_TRANSPARENT);
    } // if

    // Проверяем высоту многоугольника poly2 относительно
    // уровня моря
    if (avg_y_poly2 <= sea_level)
    {
        int ialpha = (int)((float)alpha/255 *
                          (float)(NUM_ALPHA_LEVELS-1) +
                          (float)0.5);
        // Устанавливаем альфа-цвет
        obj->plist[poly*2+1].color+= (ialpha < 24);
        // Устанавливаем в многоугольнике альфа-флаг
        SET_BIT(obj->plist[poly*2+1].attr,
                POLY4DV2_ATTR_TRANSPARENT);
    } // if
} // if
```

Для создания демонстрационной программы я взял имитатор водного мотоцикла из предыдущей главы, изменил вызов функции на новую версию 2.0 и добавил соответствующий вызов в функцию визуализации, поддерживающую альфа-канал с Z-буферизацией.

Есть еще одно свойство программы, которое немного замедляет ее работу, но оно необходимо. Когда вода прозрачна, в ней видно небо, что недопустимо. Поэтому я вынужден был поместить дополнительную сетку **ландшафта**, по сути — для создания морского дна. Поэтому, когда мы смотрим сквозь воду, мы видим морское дно, а не небо. **Про-**

грамма выполняет визуализацию двух ландшафтов, однако это не замедляет ее до неприемлемого уровня. Тем не менее, чтобы сделать программу достаточно быстрой, я вынужден был понизить разрешение до 640x480. С помощью клавиши <T> можно переключать режим отображения второго ландшафта, но я вас предупреждаю — под ним вы увидите облака! Копия экрана работающей программы DEMOII12\_4.CPP|EXE показана на рис. 12.15.

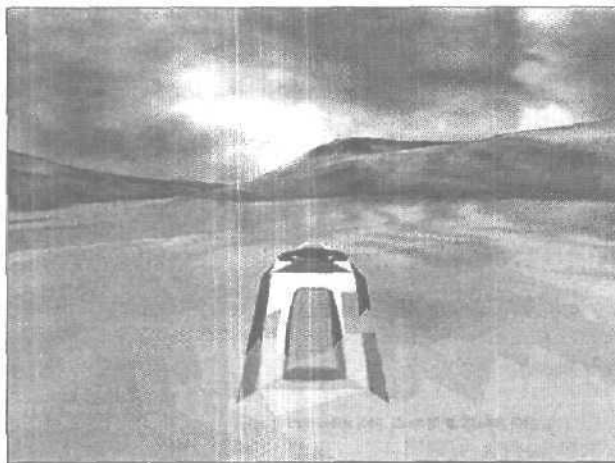


Рис. 12.15. Демонстрационная программа DEMOII12\_4.CPP|EXE с прозрачной водой

В этой демонстрационной программе клавиши управления те же, что и в предыдущей версии программы.

- <Enter> — запуск двигателя
- Стрелка вправо — поворот направо
- Стрелка влево — поворот налево
- Стрелка вверх — полный газ
- <T> — включение/выключение режима отображения морского дна
- <Z> — включение/выключение Z-буферизации
- <S> — включение/выключение Z-сортировки
- <D> — переключение между режимами нормальной визуализации/фильтра Z-буфера
- <H> — включение/выключение меню справочной системы
- <Esc> — выход из программы

Поэкспериментируйте с различными источниками света, Z-буферизацией, Z-сортировкой и т.д. Для компиляции программы, как обычно, необходимы файлы программы DEMOII12\_4.CPP|H, модули T3DLIB1-10.CPP|H и библиотечные файлы DirectX.

## Текстурирование с корректной перспективой и 1/z-буферизация

Наконец, мы готовы еще раз обратиться к вопросу корректной перспективы. Мы не могли браться за решение этой задачи, поскольку она использует ряд фундаментальных

концепций, которые мы должны были рассмотреть, такие как Z-буферизация и отображение аффинных текстур. Причина, по которой нам нужно начать с аффинного отображения текстур, очевидна, однако это не совсем так в случае Z-буферизации. В предыдущей главе мы выполнили анализ, в котором доказали, что  $1/z$ -буфер является линейным в экранном пространстве, и собирались использовать  $1/z$ -буфер вместо Z-буфера, но так и не сделали этого. По сути, мы собрали весь необходимый инструментарий не только для того, чтобы выполнить отображение текстур с корректной перспективой, но и чтобы по-настоящему понять его. Конечно, я мог бы просто дать вам информацию о системе отображения текстуры с перспективой, оптимизированной для SIMD-команд, но разве бы это способствовало пониманию вопроса?

В любом случае, в этом разделе мы собираемся рассмотреть математические принципы отображения текстур с корректной перспективой (несколькими различными способами), создать растеризаторы с  $1/z$ -буферизацией и затем объединить эти идеи для реализации системы отображения текстур с корректной перспективой (и линейно-кусочной корректной перспективой). Наконец, мы поговорим о некоторых других возможных методах аппроксимации и оптимизации. Давайте начнем с математических основ отображения текстур с корректной перспективой.

## Математические основы отображения текстур с корректной перспективой

Аффинное текстурирование, затенение по Гуро и треугольный растеризатор основаны на линейной интерполяции значений, выполняемой вдоль нисходящих ребер треугольника, подвергаемого растеризации, как показано на рис. 12.16.

Мы делали это много раз. Главное здесь то, что если вы хотите линейно интерполировать значение из вершины треугольника к его основанию, вы разбиваете его на одинаковые отрезки и интерполируете от верха к низу. Так, в общем случае, для интерполяции значения  $d$  от  $y_0$  до  $y_1$  необходимо сделать примерно следующее.

```
float d = y0;
float dy - y1 — y0;
float dxdy = d/dy;
for (y=y0; y <= y1; y++)
{
    // Делаем что-то с d

    // Интерполируем d
    d+=dxdy;
} //for
```

Проблема линейной интерполяции состоит в том, что мы не можем использовать ее для интерполяции объектов, не являющихся линейными в экранном пространстве. Следовательно, этот метод не работает для текстурирования, z-координат и даже для интенсивностей цвета. Во всех этих случаях мы искажаем данные вследствие перспективы, создаваемой с помощью координат проекции на экран. Тем не менее, все это можно исправить. Аффинные текстуры хорошо выглядят на расстоянии, неплохо выглядит затенение по Гуро с линейной интерполяцией, Z-буферизация все время показывает хорошие результаты — однако все они основаны на ошибочной логике!

На основании анализа, выполненного нами в предыдущей главе, мы доказали, что все, что мы линейно интерполируем в экранном пространстве, после выполнения аксо-

нометрического преобразования, не является линейным, и мы должны выполнить деление на  $z$ , чтобы сохранить линейность, т.е. в экранном пространстве линейны только  $1/z$ -значения. Это — ключевой момент. Я покажу вам два различных вывода **текстурирования** с корректной перспективой, которые позволят вам разобраться в данном вопросе.

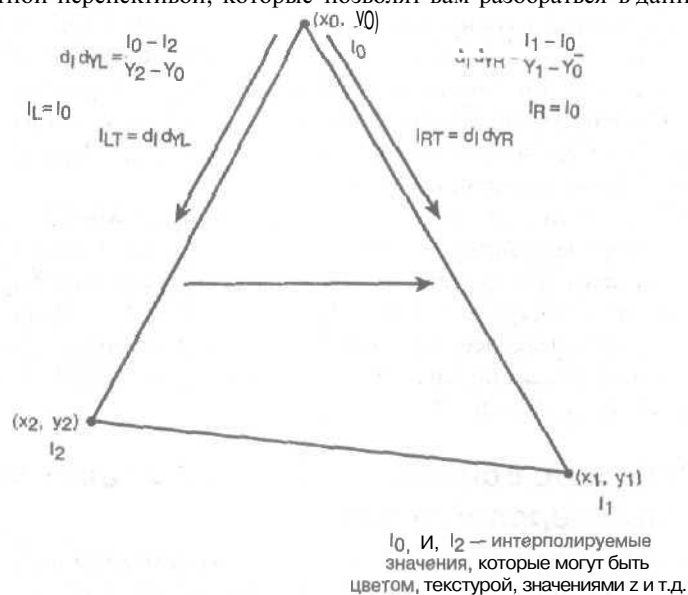


Рис. 12.16. Процесс стандартной интерполяции

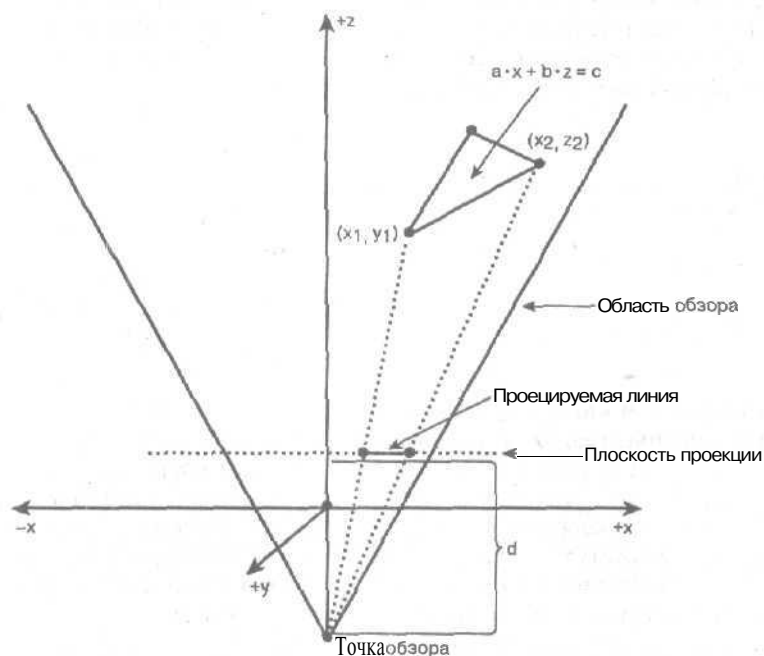


Рис. 12.17. Проецирование линии из трехмерного пространства на плоскость

## Первый вывод уравнений перспективы

На рис. 12.17 показана прямая, которую мы хотим спроецировать на экран (здесь показан двумерный случай, но трехмерный случай выглядит аналогично — просто представьте, что мы находимся в плоскости  $xz$ ,  $y=0$ ). Прямая определяется конечными точками  $(x_1, z_1)$  и  $(x_2, z_2)$ . Можно записать уравнение, которое представляет данную прямую:

$$a \cdot x + b \cdot z = c$$

Коэффициенты  $a$ ,  $b$  и  $c$  можно найти различными способами. Например, можно подставить в уравнение точки  $(x_1, z_1)$  и  $(x_2, z_2)$  и решить полученную систему уравнений:

$$a \cdot x_1 + b \cdot z_1 = c,$$

$$a \cdot x_2 + b \cdot z_2 = c.$$

Можно использовать любой из методов для решения данной системы уравнений, но сейчас нас интересуют не конкретные коэффициенты, а сама форма записи  $a \cdot x + b \cdot z = c$ .

Теперь представим, что мы проецируем точку  $(x, z)$  на прямую на экран с помощью аксонометрического преобразования. Это означает, что мы выполняем деление на  $z$  (конечно, кроме этого выполняется умножение на расстояние до наблюдателя  $d$ , но если предположить, что оно равно 1.0, то это не имеет никакого значения):

$$v = x/z.$$

Давайте запишем это уравнение иначе:

$$x = z \cdot x_{\text{пер}}$$

и подставим полученное значение  $x$  в исходное уравнение прямой:

$$a \cdot x + b \cdot z = c \rightarrow a \cdot (z \cdot x_{\text{пер}}) + b \cdot z = c.$$

Вынося  $z$  за скобки, получаем

$$(a \cdot x_{\text{пер}}) \cdot z + b \cdot z = c, \text{ или } (a \cdot x_{\text{пер}} + b) \cdot z = c.$$

Теперь мы можем найти  $z$ :

$$z = c / (a \cdot x_{\text{пер}} + b).$$

Если взять обратные значения от каждой части уравнения, то мы получим следующее.

### Уравнение 12.1. Линейное уравнение перспективы

$$1/z = (a \cdot x_{\text{пер}} + b) / c,$$

$$1/z = (a/c) \cdot x_{\text{пер}} + (b/c).$$

Это уравнение показывает, что  $1/z$  линейно зависит от  $x_{\text{пер}}$ . Если мы подставим значение  $x/z$  вместо  $x_{\text{пер}}$ , то получим:

$$1/z = (a/c) \cdot x/z + (b/c).$$

Итак,  $1/z$  линейно зависит от  $x/z$ , что, собственно, и требовалось доказать. Значит, для линейной интерполяции значения  $x$ ,  $z$ ,  $u$  и т.п. необходимо вначале разделить его на  $z$ , а уже затем — линейно интерполировать. На любом интервале интерполяции для получения исходных значений нам нужно выполнить деление на  $1/z$ . Например, мы можем подставить в уравнения координату текстуры  $u$  и посмотреть, что при этом получится:

$$1/z = (a/c) \cdot u/z + (b/c).$$

Мы начинаем с вычисления  $1/z$ , которое в дальнейшем будем называть  $z'$ . Аналогично через  $u'$  обозначим значение  $u/z$ . Затем мы выполняем один шаг линейной интерпо-

ляции. Теперь у нас есть новые значения  $u'$  и  $z'$ . Но поскольку нам нужны значения  $u$ , а не  $u'$ , мы должны выполнить деление на  $z'$ . Вот как это выглядит:

$$u = u'/z' = (u/z)/(1/z) = (u/z) \cdot (z/1) = u.$$

Итак, мы выполняем простую линейную **интерполяцию**, но на каждом шаге мы должны выполнять деление на  $z'$  для получения значения  $u$  (которое может быть чем угодно — координатой текстуры, значением **освещенности** и т.п.). Если  $u$  — это координата текстуры, то для вычисления  $u$  и  $v$  из  $u'$  и  $v'$  для каждого пикселя нам нужно выполнить две **операции** деления — это и есть цена отображения текстуры с корректной перспективой.

Прежде чем перейти к рассмотрению второго метода, давайте поговорим о том, что происходит в первом методе с точки зрения программирования. Мы **собираемся** интерполировать значения  $1/z$  по сторонам треугольника. В связи с этим мы можем без каких-либо потерь перейти от использования **Z-буфера** к использованию  **$1/z$ -буфера**. Нам просто надо инвертировать сравнение во внутреннем цикле (используя "больше" вместо "меньше") и использовать  $1/z$ -буфер. На практике надо также пересмотреть используемый формат с фиксированной точкой, чтобы получить большую точность, используя тот факт, что величина  $1/z$  не будет превышать 1.0. Нет никакой необходимости в использовании 16 бит для целой части и 16 бит для дробной части числа с фиксированной точкой. Мы должны изменить формат, например, на формат 4.28. Я вернусь к этому позже, при реализации  $1/z$ -буфера. Что же касается двух делений на каждый пиксель — то избежать их, увы, невозможно. Конечно, мы можем использовать при кодировании оптимизацию и аппроксимацию.

Перед тем как продолжить, давайте рассмотрим реальный пример и разберемся, что происходит с системой аффинного отображения текстур с корректной перспективой. Давайте выполним интерполяцию вдоль одного ребра треугольника только с одной координатой. Итак, пусть в точке  $p_1(100,100,20)$   $u_1=0$ , а в точке  $p_2(50,110,50)$   $u_2=63$  (рис. 12.18).

Ниже представлен алгоритм, который реализует аффинную/линейную интерполяцию координат текстуры.

```
float x1=100, y1=100, z1=20, u1=0,
      x2=50, y2=110, z2=50, u2=63;
```

```
float du = u2 - u1;
float dy = y2 - y1;
float dudy = du/dy;
```

```
// Инициализируем интерполянт
float u = u1;
```

```
// Интерполируем
for (int y=(int)y1; y <= (int)y2; y++)
{
    printf("\n@y=%d, u=%f", y, u);
```

```
    // Интерполируем u
    u += dudy;
} // for
```

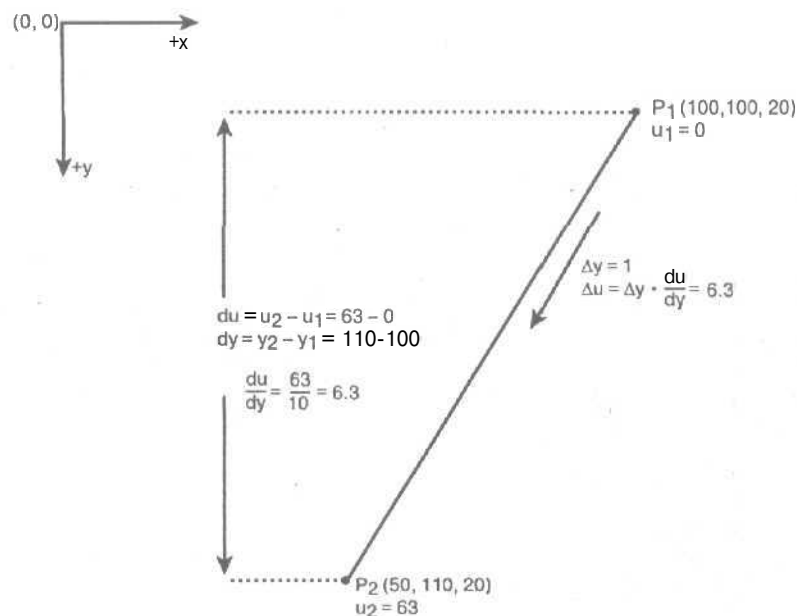


Рис. 12.18. Интерполяция перспективы

НА ЗАМЕТКУ

Получившаяся программа так мала, что я не поместил ее на компакт-диск. Если вы хотите ее запустить, просто наберите ее с клавиатуры и создайте консольное приложение.

Если вы введете и запустите эту маленькую программу, то получите следующий поток ошибочных данных, показанный в табл. 12.1.

Таблица 12.1. Линейная интерполяция координат текстуры

Координата Y	Координата текстуры
100	u = 0.000
101	u = 6.300
102	u = 12.600
103	u = 18.900
104	u = 25.200
105	u = 31.500
106	u = 37.799
107	u = 44.099
108	u = 50.399
109	u = 56.699
110	u = 62.999

Заметим, что разница между каждой итерацией в цикле одна и та же (6.3), что говорит о том, что мы игнорируем значения z. На рис. 12.19 вы можете увидеть график, построенный на основании данных из табл. 12.1.

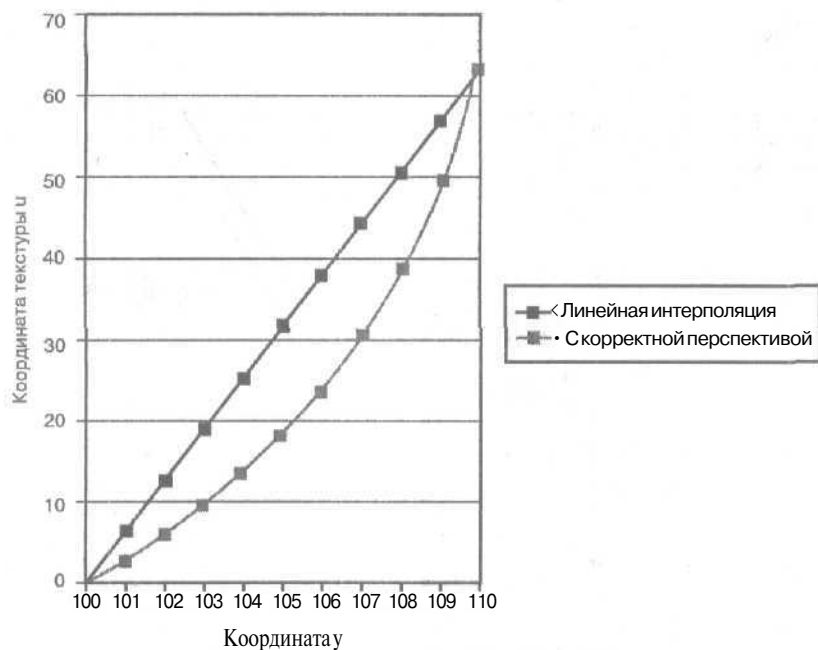


Рис. 12.19. Графики линейной и аксонометрической интерполяции

Получается, что  $u$  линейно зависит от  $z$  — **что**, как мы знаем, неверно. Давайте теперь рассмотрим на версию функции с корректной перспективой, которая интерполирует  $1/z$  и  $u/z$ , а затем в каждой итерации вычисляет  $u$  путем деления  $u/z$  на  $1/z$ .

```
float x1=100,y1=100,z1=20,u1=0,
      x2= 50,y2=110,z2=50,u2=63;

float du = u2/z2 - u1/z1;
float dy = y2 - y1;
float dz = 1/z2 - 1/z1;

float dzdy = dz/dy;
float dudy = du/dy;

// Инициализируем интерполянт
float u = u1/z1;
float z = 1/z1;

// Интерполяция
for (int y=(int)y1; y <= (int)y2; y++)
{
    printf("\n@y=%d, 1/z=%f, u/z=%f: u=%f",y, 1/z, u/z, u);
    // Интерполяция u/z
    u+=dudy;

    // Интерполяция 1/z
    z+=dzdy;
} // for
```

Теперь посмотрите на рис. 12.19 и табл. 12.2. Как видите, **интерполяция** координаты текстуры не линейна, а изменяется с изменением значения  $1/z$ , как и должно быть.

**Таблица 12.2. Интерполяция координат текстуры с корректной перспективой**

Координата $Y$	$1/z$	$u/z$	Координата текстуры
$y = 100$	$1/z = 0.000$	$u/z = 0.050$	$u = 0.000$
$y = 101$	$1/z = 0.126$	$u/z = 0.047$	$u = 2.680$
$y = 102$	$1/z = 0.252$	$u/z = 0.044$	$u = 5.727$
$y = 103$	$1/z = 0.378$	$u/z = 0.041$	$u = 9.219$
$y = 104$	$1/z = 0.504$	$u/z = 0.038$	$u = 13.263$
$y = 105$	$1/z = 0.630$	$u/z = 0.035$	$u = 17.999$
$y = 106$	$1/z = 0.756$	$u/z = 0.032$	$u = 23.624$
$y = 107$	$1/z = 0.882$	$u/z = 0.029$	$u = 30.413$
$y = 108$	$1/z = 1.008$	$u/z = 0.026$	$u = 38.769$
$y = 109$	$1/z = 1.134$	$u/z = 0.023$	$u = 49.304$
$y = 110$	$1/z = 1.26$	$u/z = 0.020$	$u = 62.999$

Для закрепления этого материала рассмотрим пример, когда координаты  $z$  точек равны. В этом случае мы не должны видеть никакой перспективы, поскольку многоугольник расположен в плоскости экрана, а рассчитанные данные должны быть идентичны данным, полученным при расчетах с использованием линейной интерполяции. Итак, чтобы убрать перспективу, зададим  $z_1 = 20$  и  $z_2 = 20$ . Результаты расчетов показаны в табл. 12.3.

**Таблица 12.3. Интерполяция координат текстуры с корректной перспективой в случае равных координат  $z$**

Координата $Y$	$1/z$	$u/z$	Координата текстуры
$y = 100$	$1/z = 0.000$	$u/z = 0.050$	$u = 0.000$
$y = 101$	$1/z = 0.315$	$u/z = 0.050$	$u = 6.300$
$y = 102$	$1/z = 0.630$	$u/z = 0.050$	$u = 12.600$
$y = 103$	$1/z = 0.945$	$u/z = 0.050$	$u = 18.900$
$y = 104$	$1/z = 1.260$	$u/z = 0.050$	$u = 25.199$
$y = 105$	$1/z = 1.575$	$u/z = 0.050$	$u = 31.500$
$y = 106$	$1/z = 1.890$	$u/z = 0.050$	$u = 37.800$
$y = 107$	$1/z = 2.205$	$u/z = 0.050$	$u = 44.100$
$y = 108$	$1/z = 2.520$	$u/z = 0.050$	$u = 50.400$
$y = 109$	$1/z = 2.835$	$u/z = 0.050$	$u = 56.700$
$y = 110$	$1/z = 3.150$	$u/z = 0.050$	$u = 63.000$

Если сравнить табл. 12.1 и 12.3, то видно, что координаты текстуры в них одинаковы (в пределах погрешности вычислений с *плавающей точкой*) — так что наш алгоритм проверен для частного случая отсутствия перспективы.

## Второй вывод уравнений перспективы

Во втором методе мы собираемся сделать нечто подобное, хотя и немного более трудоемкое, но в большей степени **соответствующее** уравнению интерполяции. Это означает немного большую строгость в окончательной формулировке уравнения двухточечной интерполяции в следующем виде.

### Уравнение 12.2. Основная формула двухточечной интерполяции

$$P = (1-t) \cdot p_1 + t \cdot p_2.$$

НА ЗАМЕТКУ

$p$  может быть как векторной, так и скалярной величиной.

Мы хотим завершить наше рассмотрение примером, который проиллюстрирует связь  $p_1$  и  $p_2$  с координатами  $z$ . Начнем со схемы на рис. 12.20.

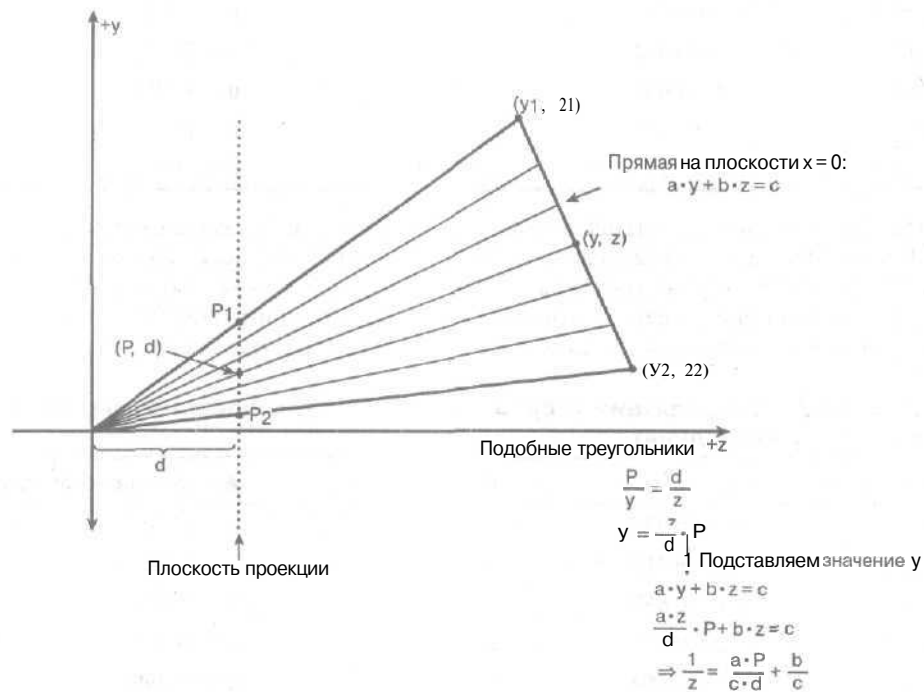


Рис. 12.20. Линейная интерполяция и аксонометрическая проекция

У нас есть две точки с мировыми координатами  $(y_1, z_1)$  и  $(y_2, z_2)$ . Эти точки проецируются на плоскость на расстоянии  $z = d$  в точки с координатами  $(p_1, d)$  и  $(p_2, d)$ . Кроме того, у нас есть произвольная точка  $(y, z)$ , которая проецируется на плоскость в точку с координатами  $(p, d)$ .

Давайте начнем с уравнения прямой, которая представляет ребро многоугольника в трехмерном пространстве, находящегося в плоскости  $yz$  при  $x = 0$ . Эта прямая описывается уравнением  $a \cdot y + b \cdot z = c$ .

Теперь, используя свойства подобных треугольников (рис. 12.20), можно установить отношение между точкой проекции  $(p, d)$  и точкой в трехмерном пространстве  $(y, z)$ :

$$p/y = d/z.$$

Отсюда находим  $y$ :

$$y = (p/d) \cdot z.$$

Подставляя  $y$  в уравнение прямой, получим

$$a \cdot y + b \cdot z = c,$$

$$a \cdot (p/d) \cdot z + b \cdot z = c,$$

$$(a \cdot p/d) \cdot z + b \cdot z = c.$$

Вынося  $z$  за скобки, имеем:

$$((a \cdot p/d) + b) \cdot z = c,$$

а выполнив деление обеих частей уравнения на выражение  $((a \cdot p/d) + b)$ , находим;

$$z = c / ((a \cdot p/d) + b).$$

Берем обратные величины от обеих частей уравнения:

$$1/z = ((a \cdot p/d) + b) / c = (a \cdot p/c \cdot d) + (b/c)$$

Вынося за скобки  $p$ , чтобы разделить постоянные и переменные величины, получим уравнение 12.3:

**Уравнение 12.3.** Обратная линейная зависимость от  $z$

$$1/z = (a/(c \cdot d)) \cdot p + (b/c).$$

Это означает, что для точки  $(y_1, z_1)$ , проекция которой  $(p_1, d)$ , можно записать уравнение

$$1/z_1 = (a/(c \cdot d)) \cdot p_1 + (b/c),$$

а для точки  $(y_2, z_2)$  спроекцией  $(p_2, d)$  —

$$1/z_2 = (a/(c \cdot d)) \cdot p_2 + (b/c).$$

Мы определили конечные точки интерполяции  $p_1$  и  $p_2$ , и теперь нам нужно посмотреть, что происходит в средней точке  $p$ . Возьмем уравнение 12.2 и подставим его в уравнение 12.3. Подставляя  $p = (1-t) \cdot p_1 + t \cdot p_2$  и  $1/2 = (a/(c \cdot d)) \cdot p + (b/c)$  получим

$$1/z = (a/(c \cdot d)) \cdot [(1-t) \cdot p_1 + (t) \cdot p_2] + (b/c).$$

Раскроем скобки:

$$1/z = (a/(c \cdot d)) \cdot p_1 \cdot (1-t) + (a/(c \cdot d)) \cdot p_2 \cdot (t) + (b/c).$$

Мы получили искомого алгебраическое выражение. Добавим к правой части и вычтем из нее  $(b/c) \cdot t$ . В результате мы получим

$$1/z = [(a/(c \cdot d) \cdot p_1) + (b/c)] \cdot (1-t) + [(a/(c \cdot d) \cdot p_2) + (b/c)] \cdot (t).$$

Теперь давайте посмотрим на члены в квадратных скобках. Они нам уже знакомы — это правые части уравнений

$$1/z_1 = (a/(c \cdot d)) \cdot p_1 + (b/c),$$

$$1/z_2 = (a/(c \cdot d)) \cdot p_2 + (b/c).$$

Подставив левые части уравнений, получаем:

$$1/z = (1/z_1) \cdot (1-t) + (1/z_2) \cdot t.$$

Уравнение показывает, что можно выполнить линейную интерполяцию от  $1/z_1$  до  $1/z_2$ . Иными словами, все, что проецируется из трехмерного пространства на экран, будет линейно, если интерполяция выполняется с делением на  $z$ .

Надеюсь, теперь вы хорошо поняли, почему  $1/z$ -интерполяция линейна и почему любое значение  $x$ , интерполируемое с делением на  $z$ , линейно в экранном пространстве. В принципе, это должно быть понятно интуитивно, поскольку для получения экранной проекции значений координат их нужно разделить на  $z$ .

## Добавление $1/z$ -буферизации в растеризаторы

Теперь, когда вы знаете причину использования величины  $1/z$  в процессе интерполяции текстур с корректной перспективой, нам нужно ввести ее поддержку в наш процессор. В этом нет ничего сложного, однако потребуются определенная работа для получения высокой точности при работе с числами с фиксированной точкой. В настоящее время мы используем формат 16.16, что позволяет использовать диапазон целых знаковых чисел от -32768 до 32767. При этом дробная часть числа может иметь наименьшее значение  $2^{-16}$ , или примерно  $1.5258 \times 10^{-5}$ , т.е. 4–5 знаков после десятичной точки. Однако все это касается формата числа самого по себе. При выполнении таких операций, как деление или умножение, эта точность может быстро снизиться. Поэтому мы вынуждены использовать другой формат числа с фиксированной точкой, который будет более удобен для работы с числами меньше 1.0 или, иными словами, с обратными значениями  $z$ . Вопрос в том, насколько существенно нужно изменить формат?

Ответ — существенно, но не слишком. Мы используем 32-битовые целые числа, и один бит нам нужен для знака числа, так что для работы остается только 31 бит. Мы можем использовать формат 2.30, в котором 1 бит будет использоваться для знака, 1 бит для "1", если числа лежат в диапазоне 0..1.99999999, но такой формат слишком "тесен". Я экспериментировал с разными форматами и остановил свой выбор на 4.28. Он означает 28 бит для дробной части, 3 бита для целой части и 1 бит — для знака. Так мы сможем записывать числа в диапазоне от -8 до +1 с 28 битами для дробной части. Этого вполне достаточно для целей  $1/z$ -буферизации и последующей интерполяции. Макроопределение для этого типа содержится в файле T3DLIB10.H.

```
#define FIXP28_SHIFT 28
```

Далее я взял растеризаторы с  $Z$ -буферизацией и преобразовал их в растеризаторы с  $1/z$ -буферизацией. После этого я написал версии, которые корректно отображают перспективу путем интерполяции аффинной текстуры по координатам  $u/z$ ,  $v/z$  с последующим делением на  $1/z$  для вычисления реальных координат текстур (вскоре мы поговорим и об этом). А пока давайте рассмотрим простейший растеризатор с  $1/z$ -буферизацией, который выводит многоугольники с постоянным/плоским затенением без использования поддержки альфа-смешивания. Ниже представлен прототип соответствующей функции.

```
void Draw_Triangle_2DINVZB_16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобуфер  
    int mem_pitch,      // Байт в строке, 320, 640 и т.д.  
    UCHAR *zbuffer,     // Указатель на Z-буфер  
    int zpitch);        // Байт в строке Z-буфера
```

Функция имеет параметры вызова, идентичные параметрам версии с использованием Z-буфера, за исключением того, что она работает с  $1/z$ -буфером. Прежде чем продолжить, давайте на некоторое время вернемся назад и вспомним, как работает  $1/z$ -буфер. Если есть пиксель с координатой  $z=100$  и другой пиксель с  $z=200$ , то пиксель с  $z=100$  должен перезаписать пиксель с  $z=200$ . Давайте посчитаем  $1/z$  для каждого из них: для пикселя с  $z=100$   $1/z=0.01$ , для пикселя с  $z=200$   $1/z=0.005$ . Поэтому нашу обычную операцию сравнения во внутреннем цикле

```
if (zi < screen_ptr[x]) { write_pixel... }
```

нужно инвертировать:

```
if (zi > screen_ptr[x]) { write_pixel... }
```

Мы уже рассматривали этот вопрос в предыдущей главе, поэтому он не должен быть новым для вас. Мы должны изменить инициализацию Z-буфера для каждого кадра, записывая в него заведомо слишком большие значения  $z$  (скажем, 32000) — при помощи кода наподобие следующего.

```
Clear_Zbuffer(&zbuffer, (32000 << FIXP16_SHIFT));
```

Однако теперь мы не только используем другой формат чисел с фиксированной точкой, но к тому же нам надо заполнить буфер нулями — т.е. мы будем использовать код

```
Clear_Zbuffer(&zbuffer, 0);
```

#### СОВЕТ

Теперь, когда мы работаем с  $1/z$ -буфером (или инверсным буфером, как я его называю), мы можем воспользоваться преимуществом уже упоминавшейся оптимизации, когда для каждого нового кадра к значению  $z$  добавляется некоторое смещение; при этом очищать буфер для вывода каждого кадра не нужно. Однако мы можем выполнять эту операцию всего лишь несколько раз подряд, поскольку в используемом формате 4.28 есть только 3 бита для хранения целых чисел. Поэтому я решил пока что не использовать эту оптимизацию.

Следующий момент — это сам код Z-буфера. Нужно ли нам что-то делать со структурами данных и т.д.? Ответ — нет, поскольку Z-буфер — это просто память. Однако мы определенно не должны смешивать вызовы Z-буфера с вызовами  $1/z$ -буфера, поскольку тогда Z-буфер превратится в "кашу".

Продолжим и рассмотрим конкретный пример кода. Ниже показан код обработки вершин в простом растеризаторе с обычной Z-буферизацией.

```
// Извлекаем координаты вершин для обработки (теперь они
// упорядочены)
```

```
x0 = (int)(face->tvlist[v0].x+0.0);
```

```
y0 = (int)(face->tvlist[v0].y+0.0);
```

```
tz0 = (int)(face->tvlist[v0].z+0.5);
```

```
x1 = (int)(face->tvlist[v1].x+0.0);
```

```
y1 = (int)(face->tvlist[v1].y+0.0);
```

```
tz1 = (int)(face->tvlist[v1].z+0.5);
```

```
x2 = (int)(face->tvlist[v2].x+0.0);
```

```
y2 = (int)(face->tvlist[v2].y+0.0);
```

```
tz2 = (int)(face->tvlist[v2].z+0.5);
```

Я выделил строки с координатами z. Теперь посмотрим, как выглядит код после ведения 1/z-буферизации.

```
// Извлекаем координаты вершин для обработки (теперь они
// упорядочены)
xO = (int)(face->tvlist[v0].x+0.0);
yO = (int)(face->tvlist[v0].y+0.0);

tz0 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v0].z+0.5);

x1 = (int)(face->tvlist[v1].x+0.0);
y1 = (int)(face->tvlist[v1].y+0.0);

tz1 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v1].z+0.5);

x2 = (int)(face->tvlist[v2].x+0.0);
y2 = (int)(face->tvlist[v2].y+0.0);

tz2 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v2].z+0.5);
```

Здесь есть два момента, которых не было в предыдущей версии. Первый: мы используем числа с фиксированной точкой. Обратите внимание, что я преобразовал число 1.0 в формат 4.28, а затем разделил его на z. Таким образом, tz0, tz1 и tz2 теперь преобразованы в формат с фиксированной точкой с самого начала, чего не было в предыдущей версии, работающей с обычной Z-буферизацией. Дело в том, что выполнять деление на z в целых числах нельзя из-за потери точности. Поэтому сначала нужно выполнить преобразование чисел в формат с фиксированной точкой, до выполнения операции деления. При этом все места кода, в которых выполнялся сдвиг с помощью константы FIXP16\_SHIFT для преобразования интерполянтов, сейчас отсутствуют, поскольку теперь числа уже находятся в формате с фиксированной точкой. С точки зрения производительности, я думаю, мы получим от этого небольшой выигрыш, но я его не подсчитывал. Итак, мы получаем числа tz0, tz1 и tz2, которые представляют собой величины 1/z в формате с фиксированной точкой 4.28, после чего растеризатор работает как обычно. Например, когда мы вычисляем все интерполянты z (я удалил здесь весь остальной код), мы обычно выполняем сдвиг всех данных с помощью значения FIXP16\_SHIFT, чтобы преобразовать их в тип с фиксированной точкой.

```
dy = (y2 - y0);
dzdyl = ((tz2 - tz0) << FIXP16_SHIFT)/dy;
dzdyr = ((tz2 - tz1) << FIXP16_SHIFT)/dy;
dy = (min_clip_y - y0);

// Вычисляем новые начальные значения LHS
zl = dzdyl*dy + (tz0 << FIXP16_SHIFT);
zr = dzdyr*dy + (tz1 << FIXP16_SHIFT);
```

Теперь 1/z-буферизация выглядит в этих местах следующим образом.

```
dy = (y2 - y0);
dzdyl = ((tz2 - tz0))/dy;
dzdyr = ((tz2 - tz1))/dy;
dy = (min_clip_y - y0);

// Вычисляем новые начальные значения LHS
```

```

zl = dzdy*dy + (tz0);
zr = dzdyr*dy + (tz1);

```

Тем самым мы избавляемся от всех сдвигов. В окончательном внутреннем цикле все остается неизменным, за исключением инвертирования сравнения для поддержки  $1/z$ -буферизации. Главное — помнить, что мы работаем с  $1/z$ -значениями, а не с  $z$ -значениями. Ниже приводится фрагмент кода функции `Draw_Triangle_2DINVZB_16()`, в котором показан цикл сканирования по координате  $y$ . Полный код функции слишком велик, здесь приведен только фрагмент, показывающий, что инверсная  $Z$ -буферизация в основном идентична обычной  $Z$ -буферизации, но при этом быстрее при прочих равных условиях.

```

// Устанавливаем указатель на начальную строку
screen_ptr = dest_buffer + (ystart * mem_pitch);

// Устанавливаем zbuffer на начальную строку
z_ptr = zbuffer + (ystart * zpitch);
for (yi = ystart; yi < yend; yi++)
{
    // Вычисляем конечные точки
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем начальные точки для интерполянтов u, v, w
    zi = zl;

    // Вычисляем интерполянты u, v
    if ((dx = (xend - xstart)) > 0)
    {
        dz = (zr - zl) / dx;
    } // if
    else
    {
        dz = (zr - zl);
    } // else

    // Выводим строку
    for (xi = xstart; xi < xend; xi++)
    {
        // Проверяем, не расположен ли текущий пиксель
        // ближе, чем текущее значение Z-буфера
        if (zi > z_ptr[xi])
        {
            // Пишем текстель в формате 5.6,5
            screen_ptr[xi] = color;

            // Обновляем Z-буфер
            z_ptr[xi] = zi;
        } // if

        // Интерполируем z
        zi += dz;
    } // for xi

    // Интерполируем x, z вдоль правого и левого ребер
    xl += dx * dy;
}

```

```

zl+=dzdyl;

xr+=dxdyr;
zr+=dzdyr;

// Обновляем указатель экрана
screen_ptr+=mem_pitch;

// Обновляем указатель Z-буфера
z_ptr+=zpitch;
} // for y

```

Давайте вкратце рассмотрим все растеризаторы для  $1/z$ -буферизации, за исключением растеризаторов с корректной перспективой. Для правильного отображения перспективы текстуры многоугольников нам нужен  $1/z$ -буфер (который работает так же, как и Z-буфер, так что мы ничего не теряем в плане производительности). Мы должны также изменить формат с фиксированной точкой для поддержки обратной величины  $z$  с достаточной точностью. Кроме того,  $1/z$ -буфер обладает лучшим разрешением для многоугольников, находящихся вблизи плоскости проекции. Таким образом, мы получаем дополнительный выигрыш в большей точности обзора на близких расстояниях. Поскольку мы изначально преобразуем все  $1/z$ -величины в формат с фиксированной точкой, мы экономим также несколько операций сдвига по сравнению с растеризатором с обычной Z-буферизацией. И наконец, мы должны очищать память  $1/z$ -буфера нулями, а не максимальными значениями  $z$  для каждого кадра, как это делалось для Z-буфера. Это дает нам возможность использовать методы сдвига буфера без его очистки при каждом кадре, что также потенциально обещает выигрыш в производительности (этот вопрос будет рассмотрен несколько позднее).

#### НА ЗАМЕТКУ

В случае растеризаторов с поддержкой альфа-смешивания нет абсолютно никакой разницы между Z-буферизацией и  $1/z$ -буферизацией. Вы просто берете растеризаторы с поддержкой альфа-смешивания и Z-буферизацией и преобразуете их в версии с инверсной Z-буферизацией. Мы познакомимся с прототипами соответствующих функций, не рассматривая их внутреннее устройство.

#### Функция

```

void Draw_Triangle_2DINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байт в строке Z-буфера

```

#### Описание

Функция `Draw_Triangle_2DINVZB_16()` выводит стандартный окрашенный треугольник с помощью инверсного Z-буфера.

#### Функция

```

void Draw_Gouraud_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байт в строке Z-буфера

```

### Описание

Функция `Draw_Gouraud_TriangleINVZB_16()` выводит стандартный треугольник с затенением по Гуро с помощью инверсного Z-буфера.

### Функция

```
void Draw_Textured_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int mem_pitch);      // Байтов в строке, 320, 640 и т.д.
```

### Описание

Функция `Draw_Textured_TriangleINVZB_16()` выводит треугольник с аффинным текстурованием и постоянным затенением с помощью инверсного Z-буфера.

### Функция

```
void Draw_Textured_TriangleFSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_Textured_TriangleFSINVZB_16()` выводит треугольник с аффинным текстурованием и плоским затенением с помощью инверсного Z-буфера.

### Функция

```
void Draw_Textured_TriangleGSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_Textured_TriangleGSINVZB_16()` выводит треугольник с аффинным текстурованием и затенением по Гуро с помощью инверсного Z-буфера.

### Функция

```
void Draw_RENDERLIST4DV2_SolidINVZB_16(
    RENDERLIST4DV2_PTR rendjist
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int lpitch,          // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_RENDERLIST4DV2_SolidINVZB_16()` — это функция визуализации списка, которая вызывает растеризаторы, основываясь на информации о затенении и текстуровании треугольника. Функция вызывает только функции с инверсным Z-буфером.

Далее идут растеризаторы с поддержкой альфа-смешивания и использованием инверсного Z-буфера. Они идентичны растеризаторам без поддержки альфа-смешивания, за исклю-

чением переменной `alpha_override`, добавляемой к списку параметров (я не упоминаю версии с использованием перспективы, поскольку мы пока что не рассматривали эту тему).

#### Функция

```
void Draw_Triangle_2DINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
int alpha);             // Уровень альфа
```

#### Описание

Функция `Draw_Triangle_2DINVZB_Alpha16()` выводит стандартный окрашенный треугольник с помощью инверсного Z-буфера.

#### Функция

```
void Draw_Gouraud_TriangleINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
int alpha);             // Уровень альфа
```

#### Описание

Функция `Draw_Gouraud_TriangleINVZB_Alpha16()` выводит стандартный треугольник с затенением по Гуро с помощью инверсного Z-буфера.

#### Функция

```
void Draw_Textured_TriangleINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
int alpha);             // Уровень альфа
```

#### Описание

Функция `Draw_Textured_TriangleINVZB_Alpha16()` выводит треугольник с аффинным текстурированием и постоянным затенением с помощью инверсного Z-буфера.

#### Функция

```
void Draw_Textured_TriangleFSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,      // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
int alpha);             // Уровень альфа
```

#### Описание

Функция `Draw_Textured_TriangleFSINVZB_Alpha16()` выводит треугольник с аффинным текстурированием и плоским затенением с помощью инверсного Z-буфера.

### Функция

```
void Draw_Textured_TriangleGSINVZB_Alpha16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.  
    UCHAR *zbuffer,     // Указатель на Z-буфер  
    int zpitch);        // Байтов в строке Z-буфера  
int alpha);            // Уровень альфа
```

### Описание

Функция `Draw_Textured_TriangleGSINVZB_Alpha16()` выводит треугольник с аффинным текстурированием и затенением по Гуро с помощью инверсного Z-буфера.

### Функция

```
void Draw_RENDERLIST4DV2_SolidINVZB_Alpha16(  
    RENDERLIST4DV2_PTR rend_list,  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.  
    UCHAR *zbuffer,     // Указатель на Z-буфер  
    int zpitch);        // Байтов в строке Z-буфера  
int alpha);            // Уровень альфа
```

### Описание

Функция `Draw_RENDERLIST4DV2_SolidINVZB_Alpha16()` — это функция визуализации списка, которая вызывает растеризаторы, основываясь на информации о затенении и текстурировании треугольника. Функция вызывает только функции с инверсным Z-буфером.

На рис. 12.21 показана копия экрана демонстрационной программы `DEMO112_5.CPP EXE`, которая использует 1/z-буферизацию.



Рис. 12.21. Копия экрана демонстрационной программы, использующей технологию 1/z-буферизации

Эта программа позволяет переключаться между режимами использования Z-буфера, 1/z -буфера и без использования буфера с помощью клавиши <Z>. Можно также включать и выключать Z-сортировку с помощью клавиши <S>. Поэкспериментируйте с объектами в различных режимах буферизации — какой из них, по-вашему, более точно выполняет визуализацию? Для компиляции этой программы вам нужны файлы DEMOII12\_5.CPP|H, T3DLIB1-10.CPP|H и библиотечные файлы DirectX.

**СОВЕТ**

Попробуйте отключить источник рассеянного света с помощью клавиши <A> — при этом сцена выглядит особенно впечатляюще.

## Реализация отображения сточной перспективой

На данной стадии корректное отображение текстур требует только небольшого редактирования исходных текстов. Начнем с растеризатора аффинной текстуры с Z-буферизацией и внесем некоторые изменения в интерполянты и окончательное вычисление координат  $u$ ,  $v$  для каждого пикселя. Изменения вносятся в следующие разделы.

- В процессе инициализации нам нужно вычислить координаты текстуры  $u/z$  и  $v/z$  и интерполировать их наряду с  $1/z$ .
- Во внутреннем цикле растеризатора нам нужно вычислить реальные координаты текстуры  $u$  и  $v$ . Это выполняется путем деления линейно интерполированных значений  $u/z$  и  $v/z$  на  $1/z$ .

Конечно, трудность любой проблемы заключена в ее деталях. Прежде всего, мы должны еще раз вернуться к формату с фиксированной точкой. Главная проблема состоит в том, что мы больше не интерполируем значения  $u$  и  $v$ , мы интерполируем значения  $u/z$  и  $v/z$ , поэтому нам может понадобиться более 16 бит для дробной части. Однако мы не можем использовать формат 4.28, который мы создали для интерполяции  $1/z$ , поскольку сами координаты текстуры могут достигать значения 255. Для диапазона 0..255 нам нужно не менее 8 бит, т.е. нам нужно 9 бит, но так как я ненавижу нечетные числа, нам нужно 10 бит для целочисленной части текстурной координаты. Следовательно, для дробной части остается 22 бита.

Итак, для интерполяции значений  $u/z$  и  $v/z$  мы будем использовать числа с фиксированной точкой в формате 10.22, а для интерполяции значений  $1/z$  — формат 4.28. Это означает, что, когда в процессе окончательного деления  $u/z$  и  $v/z$  на  $1/z$  мы будем делить число с фиксированной точкой формата 10.22 на число с фиксированной точкой формата 4.28, так что нам надо выполнить правильный сдвиг разрядов чисел. Что ж, будем помнить об этом. Посмотрим на код для вершин и координат  $u$ ,  $v$ . Ниже приводится фрагмент кода растеризатора текстур с корректной перспективой, постоянным затенением на основе инверсной Z-буферизации.

```
// Извлекаем координаты вершин для обработки (теперь они
// упорядочены)
x0 = (int)(face->tvlist[v0].x+0.0);
y0 = (int)(face->tvlist[v0].y+0.0);

tu0 = ((int)(face->tvlist[v0].u0+0.5) << FIXP22_SHIFT) /
      (int)(face->tvlist[v0].z+0.5);
tv0 = ((int)(face->tvlist[v0].v0+0.5) << FIXP22_SHIFT) /
      (int)(face->tvlist[v0].z+0.5);
```

```
tz0 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v0].z+0.5);
```

```
x1 = (int)(face->tvlist[v1].x+0.0);
```

```
y1 = (int)(face->tvlist[v1].y+0.0);
```

```
tu1 = ((int)(face->tvlist[v1].u0+0.5) << FIXP22_SHIFT) /  
      (int)(face->tvlist[v1].z+0.5);
```

```
tv1 = ((int)(face->tvlist[v1].v0+0.5) << FIXP22_SHIFT) /  
      (int)(face->tvlist[v1].z+0.5);
```

```
tz1 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v1].z+0.5);
```

```
x2 = (int)(face->tvlist[v2].x+0.0);
```

```
y2 = (int)(face->tvlist[v2].y+0.0);
```

```
tu2 = ((int)(face->tvlist[v2].u0+0.5) << FIXP22_SHIFT) /  
      (int)(face->tvlist[v2].z+0.5);
```

```
tv2 = ((int)(face->tvlist[v2].v0+0.5) << FIXP22_SHIFT) /  
      (int)(face->tvlist[v2].z+0.5);
```

```
tz2 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v2].z+0.5);
```

Само собой, константа `FIXP22_SHIFT` определена как

```
#define FIXP22_SHIFT 22
```

Постарайтесь хорошо уяснить, чего мы хотим добиться при помощи этого кода. Мы делаем то же самое, что и при  $1/z$ -вычислениях, однако теперь координаты текстуры  $(u, v)$  для каждой вершины вычисляются как  $(u/z, v/z)$ . Вначале мы выполняем сдвиг значений  $u, v$  для преобразования их в формат 10.22, а затем делим на  $z$ . Однако мы уже доказали, что число с фиксированной точкой, умноженное или деленное на целое число, остается числом с фиксированной точкой — конечно, при условии, что не происходит переполнения или потери значащих разрядов числа. Следовательно, в конце представленной инициализации у нас есть значения  $1/z$  и все координаты текстуры для каждой вершины в формате  $(u/z, v/z)$ , готовые к линейной интерполяции. Поверьте мне на слово — код внутреннего цикла растеризатора, в котором выполняются основные вычисления, остается без изменений. Однако нам нужны реальные координаты текстуры  $u$  и  $v$ , для получения которых каждый интерполянт  $u/z$  и  $v/z$  нужно делить на  $1/z$ :

$$u = (u/z) / (1/z),$$
$$v = (v/z) / (1/z).$$

Однако  $u/z$  и  $v/z$  — это числа с фиксированной точкой в формате 10.22, а  $1/z$  — это число с фиксированной точкой в формате 4.28, поэтому непосредственное деление здесь не подходит. Перед делением нужно выполнить сдвиг одного или обоих операндов. Ниже показан код, реализующий это действие.

```
for (xi=xstart; xi < xend; xi++)
```

```
{
```

```
    // Проверяем, не находится ли текущий пиксель ближе к  
    // наблюдателю, чем находящийся в Z-буфере
```

```
    if (zi > z_ptr[xi])
```

```
    {
```

```
        // Записываем текстель
```

```
        screen_ptr[xi] =
```

```
            textmap[((ui << (FIXP28_SHIFT-FIXP22_SHIFT))/zi)+
```

```

        (((vi << (FIXP28_SHIFT - FIXP22_SHIFT)) / zi)
        << texture_shift2));

    // Обновляем Z-буфер
    z_ptr[xi] = zi;
} // if

// Интерполируем u, v, z
ui += du;
vi += dv;
zi += dz;
} // for xi

```

Я выделил "магический" код, который при обычном аффинном текстурировании выглядит следующим образом.

```

screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
((vi >> FIXP16_SHIFT) << texture_shift2)];

```

Как видите, в нем появилось деление на  $z_i$  и сдвиг на величину  $(\text{FIXP28\_SHIFT} - \text{FIXP22\_SHIFT})$ . Давайте выясним, почему она имеет именно такое значение. Взгляните на рисунок 12.22.

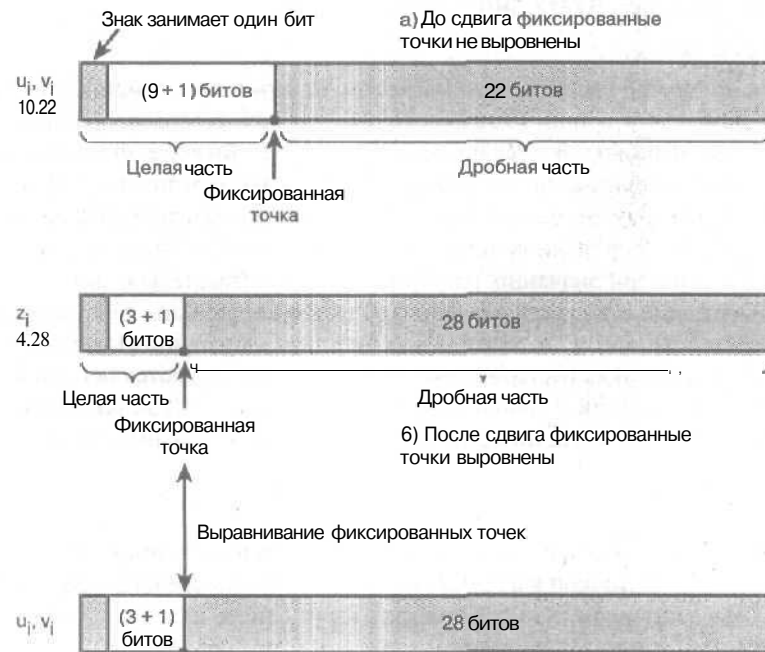


Рис. 12.22. Выравнивание фиксированных точек

Значения  $u_i$  и  $v_i$  находятся в формате 10.22, а  $z_i$  — в формате 4.28. Следовательно, нам нужно выровнять фиксированные точки, чтобы результат деления не требовал масштабирования и был, таким образом, целым. Чтобы фиксированную точку  $u_i$  и  $v_i$  выровнять с фиксированной точкой  $z_i$ , эти числа нужно сдвинуть на шесть разрядов влево. Это связано с тем, что указанные два формата отличаются ровно на

$$(\text{FIXP28\_SHIFT} - \text{FIXP22\_SHIFT}) = (28 - 22) = 6 \text{ битов}$$

Следовательно, после сдвига и делимое, и делитель **находятся** в формате 4.28, а частное не масштабируется и остается целым.

**СОВЕТ**

Есть еще одна потенциальная возможность оптимизации, связанная с адресацией текстур. Если посмотреть код, в нем выполняется следующее обращение к текстуре:

```
texture[u + (v << n)]
```

Здесь  $n$  — это  $\log_2$  от ширины текстуры (или количества слов в строке), т.е. по сути сдвиг эквивалентен умножению на значение ширины текстуры. Если выполнить предвычисление масштабированного значения  $vi$ , то использования сдвига в окончательном расчете адреса текстуры можно избежать и тем самым увеличить скорость вычисления адреса текстуры примерно на 30%. Позднее мы еще вернемся к этому моменту.

Конечно, два деления на один пиксель — это неприемлемо, и очень скоро мы предложим ряд решений этой проблемы. По большому счету, деление всегда можно превратить в умножение путем вычисления обратной к делителю величины. Например, рассмотрим следующий код.

```
float x,y,c;  
// Некоторые вычисления с величинами x, y, c...  
x = x/c;  
y = y/c;
```

Исходя из того, что одно деление занимает до 15–40 тактов процессора, выполнение данного кода требует 30–80 тактов. Теперь давайте выполним следующее простое преобразование.

```
float x,y,c;  
// Некоторые вычисления с величинами x, y, c...  
float oneoverc = 1/c;  
x = x*oneoverc;  
y = y*oneoverc;
```

Теперь у нас есть одна операция деления (15–40 тактов) и два умножения (по 2–3 такта каждое), итого — 34–46 тактов (конечно, моя оценка достаточно пессимистична). Итак, мы получаем практически 50-процентную экономию процессорного времени. Любое деление во внутреннем цикле вообще неприемлемо — нужно обязательно искать другие подходы. Наши алгоритмические возможности связаны с аппроксимацией и мы позднее рассмотрим их; а пока давайте рассмотрим новые функции, поддерживающие текстуры с корректной перспективой с помощью использования инверсного Z-буфера.

**Функция**

```
void Draw_Textured_Perspective_Triangle_INVZB_Alpha16(  
    POLYF4DV2_PTR face, // Указатель на треугольник  
    UCHAR *_dest_buffer, // Указатель на видеобuffer  
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.  
    UCHAR *_zbuffer,    // Указатель на Z-буфер  
    int zpitch,         // Байтов в строке Z-буфера  
    int alpha);         // Уровень альфа
```

**Описание**

Функция `Draw_Textured_Perspective_Triangle_INVZB_Alpha16()` выводит треугольник с корректной перспективой с поддержкой альфа-смешивания и постоянным затенением,

**Функция**

```
void Draw_Textured_Perspective_Triangle_FSINVZB_Alpha16(  
    POLYF4DV2_PTR face, // Указатель на треугольник
```

```

    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера
    int alpha);         // Уровень альфа

```

#### Описание

Функция `Draw_Textured_Perspective_Triangle_FSINVZB_Alpha16()` выводит треугольник с корректной перспективой с поддержкой альфа-смешивания и плоским затенением.

#### Функция

```

void Draw_Textured_Perspective_Triangle_INVZB_16(
    POLYF4DV2_PTRface, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера

```

#### Описание

Функция `Draw_Textured_Perspective_Triangle_INVZB_16()` выводит треугольник с корректной перспективой и постоянным затенением.

#### Функция

```

void Draw_Textured_Perspective_Triangle_FSINVZB_16(
    POLYF4DV2_PTRface, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch,         // Байтов в строке Z-буфера

```

#### Описание

Функция `Draw_Textured_Perspective_Triangle_FSINVZB_16()` выводит треугольник с корректной перспективой и плоским затенением.

В качестве демонстрационной программы **текстурирования** с корректной перспективой лучше всего было бы выбрать показ интерьера **помещения**, однако у нас пока что нет хороших инструментов для построения интерьеров. Вместо этого демонстрационная программа `DEMOII12_6.CPP|EXE` строит плоский ландшафт с текстурой шахматной доски. Как видите, различие между аффинной текстурой и текстурой с корректной перспективой довольно значительно. Копия экрана программы приведена на рис. 12.23.

Для перемещения в пространстве используйте клавиши управления курсором, а с помощью клавиши <T> можно переключаться между аффинным текстурированием и текстурированием с корректной перспективой. Для компиляции демонстрационной программы вам потребуются файлы `DEMOII12_6.CPP|H`, `T3DLIB1-10.CPP|H`, а также библиотечные файлы DirectX.

## Отображение текстуры с линейно-кусочной перспективой

Хотя мы можем ускорить отображение совершенной текстуры с помощью трюка замены деления умножением на обратную величину, деление есть деление и в большинстве случаев оно неприемлемо для игрового процессора. Что же можно сделать в этой ситуации? Можно ли создать аппроксимацию, работающую быстрее и дающую более привлекательно **выглядящий** результат? Ответ — все зависит от обстоятельств.

Press ESC to exit. Press GH for Help.

CAM [320.72, 64.00, 204.95], CELL [1, 1, 1]  
 Map: [C:\Program Files\Giga-Byte\Polys lit: 1678]  
 Options[0/4]: Ambient=1, Infinite=1, Paint=1, FastFwMCR, Terrain: MODE [PERSPECTIVE CORRECT]

Аппроксимация, которую мы собираемся реализовать, называется линейно-кусочной (или кусочно-линейной, что одно и то же). В целом, это что-то среднее между аффинным **текстурированием** и текстурированием с корректной перспективой (рис. 12.24).

Мы вычисляем координаты правильной текстуры вдоль кривой перспективы, а затем выполняем интерполяцию между ними. По мере того, как число точек выборки растет, полученная аппроксимация все более и более приближается к кривой текстуры с точной перспективой. Следовательно, по сути это технология для текстурирования с заданной степенью точности. Контролируя качество и точность **визуализации**, мы сможем экономить время выполнения визуализации за счет уменьшения числа операций деления на один пиксель путем обработки каждого  $n$ -го пикселя с последующей интерполяцией между ними. При приближении **объектов** можно увеличить число точек выборки, а при удалении многоугольников от плоскости наблюдения можно уменьшить их **число**, и текстурирование станет близким к аффинному.

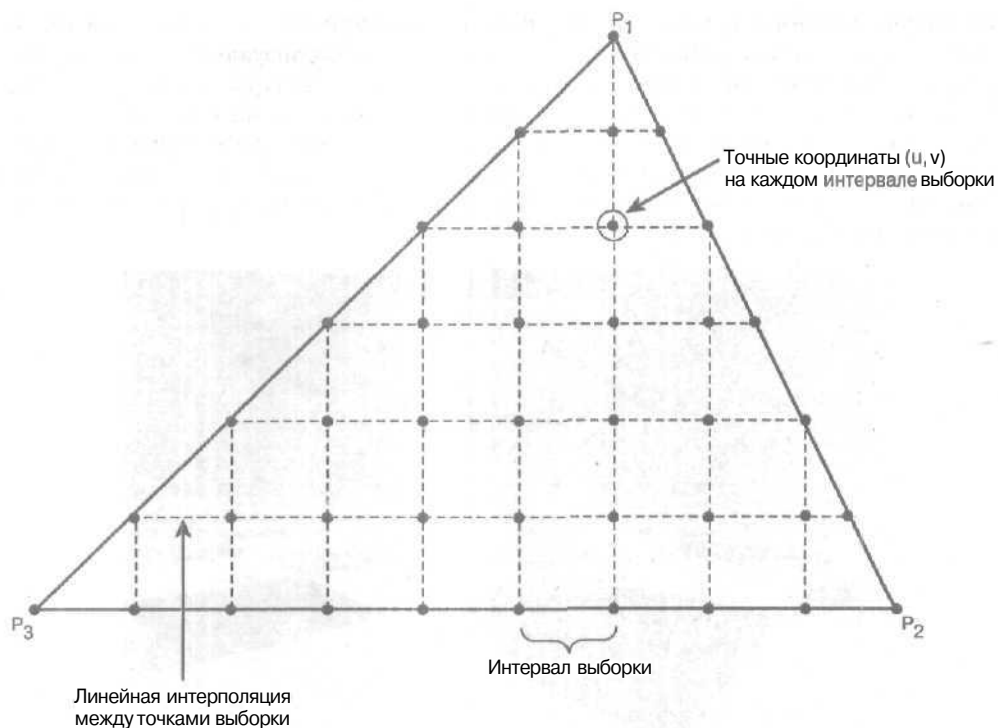


Рис. 12.24, Точки выборки линейно-кусочной аппроксимации в треугольнике

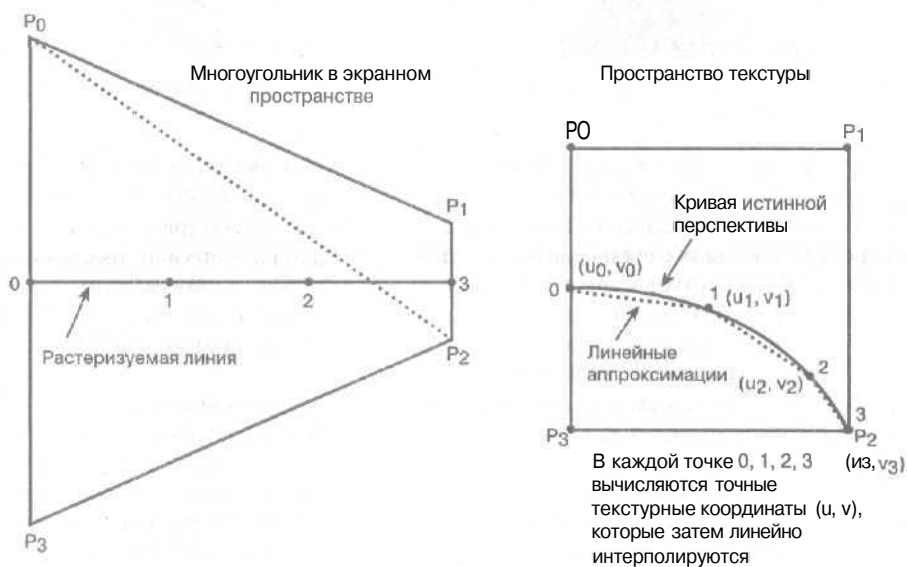


Рис. 12.25. Кривая перспективы и линейно-кусочная аппроксимация

На практике я обнаружил, что вычисление координат текстуры для каждого 16-32 пикселя выглядит почти идеально, даже когда текстура максимально приближена к на-

блюдателю, А теперь о неприятностях: реализация алгоритма весьма запутана; написание кода для обработки каждого  $p$ -ГО пикселя представляет собой непростую задачу. Имеется несколько способов реализации кусочной выборки, показанных на рис. 12.26 и перечисленных ниже.



Рис. 12.26. Методы вычисления координат текстуры для линейно-кусочной текстуры

- **Метод I.** Точные координаты текстуры вычисляются для каждой  $p$ -й строки раstra и каждого  $p$ -го пикселя вдоль каждой строки раstra.
- **Метод II.** Точные координаты текстуры вычисляются для каждой строки раstra и каждого  $p$ -го пикселя вдоль этой строки.
- **Метод III.** Точные координаты текстуры вычисляются для каждой строки раstra, а затем строка раstra линейно интерполируется.

Сейчас мы собираемся реализовать метод 3. Метод 2 мы можем реализовать позже. Это связано с тем, что у нас уже есть алгоритм вычисления точных координат текстуры в каждой строке раstra, реализованный в только что написанной программе точного растеризатора текстуры. Мы можем просто отключить растеризацию и линейно интерполировать каждую строку раstra с точными координатами текстуры  $u, v$  на концах отрезка. Мы могли бы реализовать метод 2, но для него необходимы большие затраты вычислительных ресурсов в связи с вычислением координат  $u, v$  для каждого  $p$ -го пикселя строки раstra, что в данный момент нам не требуется. Мы вернемся к этому методу, когда потребуется выполнение визуализации интерьеров.

Итак, наш план таков: взять растеризатор текстуры с корректной перспективой и изменить его так, чтобы вместо вычисления точных координат  $u, v$  для каждого пикселя мы вычисляли точные координаты  $u, v$  только в конечных точках каждой строки раstra. Затем, используя точные координаты  $u, v$  конечных точек строки раstra, мы можем выполнить интерполяцию в пределах строки слева направо.

Теперь посмотрим на код, решающий поставленную задачу. Ниже приводится фрагмент кода простейшего линейно-кусочного растеризатора текстуры `Draw_Textured_PerspectiveLP_Triangle_INVZB_16()`. Ниже представлен основной цикл, выполняющий визуализацию треугольника.

```
// Устанавливаем указатель экрана на начальную строку
screen_ptr = dest_buffer + (ystart * mem_pitch);
```

```

// Устанавливаем zbuffer на начальную строку
z_ptr - zbuffer + (ystart * zpitch);
for {yi - ystart; yi < yend; yi++}
{
    // Вычисляем конечные точки
    xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
    xend - ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

    // Вычисляем линейную версию ul, ur, vl, vr
    ul2 = ((ul << (FIXP28_SHIFT - FIXP22_SHIFT)) /
        (zl >> 6) ) << 16;
    ur2 = ((ur << (FIXP28_SHIFT - FIXP22_SHIFT)) /
        (zr >> 6) ) << 16;
    vl2 = ((vl << (FIXP28_SHIFT - FIXP22_SHIFT)) /
        (zl >> 6) ) << 16;
    vr2 = ((vr << (FIXP28_SHIFT - FIXP22_SHIFT)) /
        (zr >> 6) ) << 16;

    // Вычисляем начальные точки для интерполянтов u, v
    zi - zl + 0;
    ui - ul2 + 0;
    vi - vl2 + 0;

    // Вычисляем интерполянты u, v
    if ((dx = (xend - xstart)) > 0)
    {
        du - (ur2 - ul2) / dx;
        dv - (vr2 - vl2) / dx;
        dz = (zr - zl) / dx;
    } // if
    else
    {
        du = (ur2 - ul2) ;
        dv = (vr2 - vl2) ;
        dz = (zr - zl);
    } // else

    // Выводим строку
    for (xi=xstart; xi < xend; xi++)
    {
        // Проверяем, не располагается ли текущий пиксель
        // ближе к наблюдателю по координате z, чем уже
        // находящийся в буфере
        if (zi > z_ptr[xi])
        {
            // Записываем текстель
            screen_ptr[xi] - textmap[(ui >> FIXP22_SHIFT) +
                ((vi >> FIXP22_SHIFT) << texture_shift2)];
            // Обновляем Z-буфер
            z_ptr[xi] = zi;
        } // if
    }
}

```

```

// Интерполируем u, v, z
ui+=du;
vi+=dv;
zi+=dz;
} // for xi

// Интерполируем u, v, x вдоль правого и левого ребер
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyL;
zl+= dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
zr+=dzdyr;

// Перемещаем указатель экрана
screen_ptr+=mem_pitch;

// Перемещаем указатель Z-буфера
z_ptr+=zpitch;
} // for yi

```

Я знаю, что трудно разобраться в коде без контекста, но все же постарайтесь это сделать. Я выделил два основных фрагмента, в которых происходит действие. В первом из них вычисляются точные координаты  $u$ ,  $v$  строки сканирования путем деления на  $1/z$ . Результаты сохраняются в переменных  $ul2$ ,  $vl2$ ,  $ur2$  и  $vr2$ . Остальное тривиально. Вводится внутренний цикл растеризации (второй выделенный фрагмент), и значения координат текстуры ( $ul2$ ,  $vl2$ ) и ( $ur2$ ,  $vr2$ ) линейно интерполируются. Хотя мы все еще используем деление для вычисления точных координат текстуры  $u$ ,  $v$ , мы делаем это для отдельной строки раstra, а не для отдельного пикселя.

Это все, что касается отображения текстуры с линейно-кусочной перспективой. Оно представляет собой не более, чем периодическое вычисление точных координат текстуры с последующей линейной интерполяцией между ними. Давайте рассмотрим новые функции, которые реализуют эти возможности (заметим, что все эти функции работают только с инверсным Z-буфером).

#### Функция

```

void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

```

#### Описание

Функция `Draw_Textured_PerspectiveLP_Triangle_FSINVZB_16()` выводит текстуру с линейно-кусочной корректной перспективой и плоским затенением.

#### Функция

```

void Draw_Textured_PerspectiveLP_Triangle_INVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник

```

```

    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера

```

#### Описание

Функция `Draw_Textured_PerspectiveLP_Triangle_INVZB_16()` выводит текстуру с линейно-кусочной корректной перспективой и постоянным затенением.

#### Функция

```

void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch);         // Байтов в строке Z-буфера
int alpha);             // Альфа

```

#### Описание

Функция `Draw_Textured_PerspectiveLP_Triangle_FSINVZB_Alpha16()` выводит текстуру с линейно-кусочной корректной перспективой, плоским затенением и альфа-смешиванием.

#### Функция

```

void Draw_Textured_PerspectiveLP_Triangle_INVZB_Alpha16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,       // Байтов в строке 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch,          // Байтов в строке Z-буфера
    int alpha);          // Альфа

```

#### Описание

Функция `Draw_Textured_PerspectiveLP_Triangle_INVZB_Alpha16()` выводит текстуру с линейно-кусочной корректной перспективой, постоянным затенением и альфа-смешиванием.

#### НА ЗАМЕТКУ

Я прошу прощения за столь длинные имена этих функций, однако у меня нет другого выхода. Было бы значительно хуже, если бы я начал вводить в них такие слова, как *inverse*, *Z-buffer*, *Gouraud* и т.п.

Для сравнения аффинного, линейно-кусочного **текстурирования** и **текстурирования** с точной перспективой я изменил предыдущую демонстрационную программу, внося в нее поддержку **линейно-кусочной** растеризации. На рис. 12.27 показана копия экрана этой программы `DEMO1112_7.CPP|EXE`.

Как видите, все это выглядит вполне приемлемо. Клавиши управления те же — перемещение с **помощью** клавиш управления курсором, клавиша <T> обеспечивает переключение между режимами аффинного, линейного и идеального текстурирования.

## Квадратичные аппроксимации для текстурирования с перспективой

Еще один метод аппроксимации, которой мы обсудим, называется квадратичной аппроксимацией. Он основан на идее **линейно-кусочной** аппроксимации, однако вместо линейной интерполяции каждой строки раstra для аппроксимации кривой перспективы здесь используется квадратичная кривая. Сейчас вы, наверное, подумали, что это еще хуже, чем два деления на пиксель, но это не так! По сути, это так же легко реализуется, как и линейно-кусочный метод, с тем отличием, что данный метод почти идентичен методу текстурирования с точной перспективой. Скажу сразу, что мы не собираемся реализовать этот метод, поскольку он нам не нужен (но при желании вы можете сделать это самостоятельно). Тем не менее, мы подробно рассмотрим механизм его работы (который, как оказывается, довольно прост).

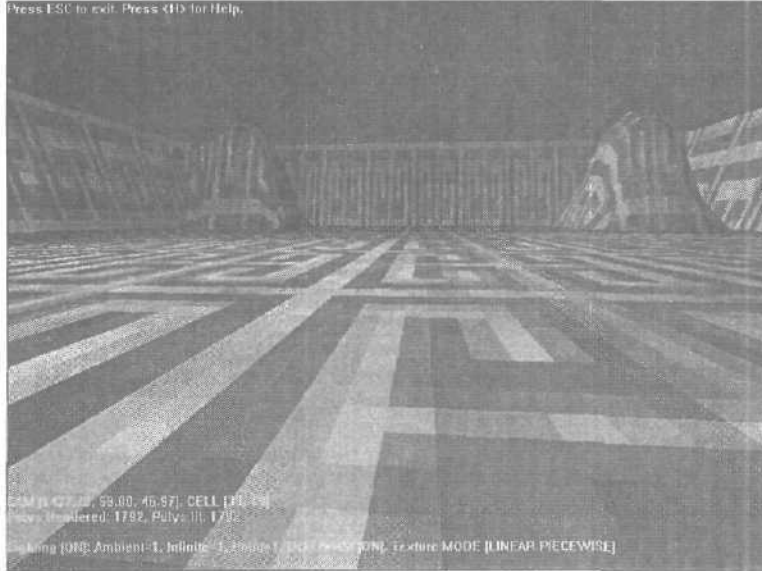


Рис. 12.27. Линейно-кусочное текстурирование в действии

Вначале сформулируем задачу. Предположим, мы вычисляем точные координаты текстуры  $u, v$  строки раstra (возможно, с помощью замены операции деления операцией умножения на обратную величину). Мы хотим достичь лучшей аппроксимации вдоль строки раstra, чем та, что получается при линейной аффинной интерполяции, однако мы не хотим прибегать к разбиению строки на большое число отрезков. Решение состоит в интерполяции строки раstra слева направо не линейно, а с помощью кривой, которая проходит через три точки истинной кривой перспективы со значительно большим приближением к ней, чем в случае с прямой линией. Рис. 12.28 схематически показывает, чего мы добиваемся.

Словом, нам нужно получить приближение к кривой вида  $c/z$ , где  $c$  — это константа, и получить его мы хотим при помощи использования квадратичной функции вида  $f(x) = ax^2 + bx + c$ .

График такой функции называется параболой. Например, на рис. 12.29 показана парабола  $f(x) = 1 \cdot x^2 + 0 \cdot x + 3$ .

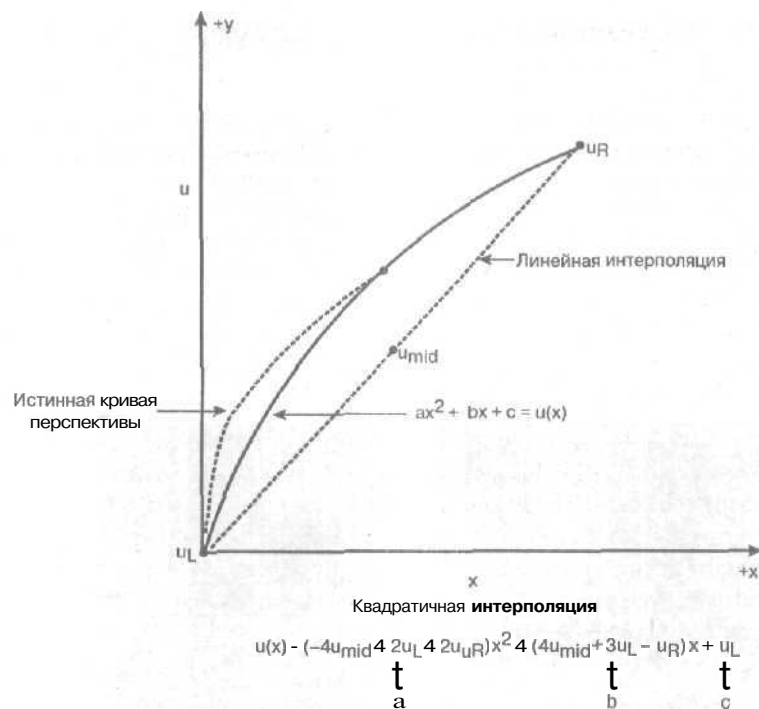


Рис. 12.28. Квадратичная аппроксимация кривой перспективы

Нам нужно найти способ подобрать коэффициенты в формуле  $ax^2 + bx + c$ , чтобы они давали приближение интересующей нас кривой  $c/z$ .

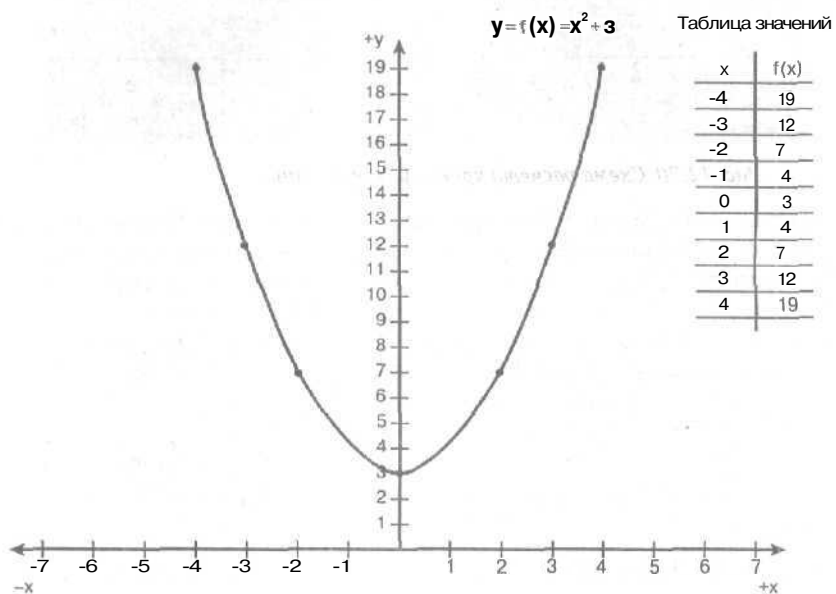


Рис. 12.29. Стандартная парабола

Выбор правильных коэффициентов позволил бы нам максимально приблизиться к точной кривой перспективы. Мы воспользуемся таким **приемом** — вычислим координаты текстуры на концах строки раstra, а затем используем их для поиска коэффициентов  $a$ ,  $b$  и  $c$ . Тогда интерполяционная кривая обязательно пройдет через конечные точки, но как поступить в средней части кривой? Чтобы ответить на этот вопрос, нужна дополнительная информация. У нас **есть** три неизвестных ( $a$ ,  $b$  и  $c$ ) и только две известных величины ( $u_l$ ,  $u_r$  (или  $v_l$ ,  $v_r$ )), являющихся координатами текстуры для конечных точек, между которыми мы пытаемся выполнять интерполяцию (и  $u$  и  $v$  независимы, и должны интерполироваться отдельно, так что это две разные кривые). Нам нужна дополнительная информация.

Такую дополнительную информацию можно получить разными способами. Например, можно потребовать, чтобы наклон точной кривой и кривой интерполяции совпадали в начальной точке отрезка (т.е. совпадали значения производных), можно потребовать такого совпадения в конечной точке — **словом**, варианты могут быть самыми разными. Мы же рассмотрим более простое требование — совпадения интерполяционной кривой с точной в середине интерполируемого отрезка (рис. 12.30). Для простоты вычислений будем считать, что координаты  $x$  концов отрезка интерполяции равны соответственно 0 и 1, а середины — 0.5 (переход к другим значениям не должен представлять для вас какой-либо трудности). Тогда наша задача сводится к поиску точного значения координаты текстуры в средней точке  $u_{mid}$ . Мы уже знаем, как искать данное значение — путем линейной интерполяции величин  $u/z$  ( $v/z$ ) и  $1/z$  с последующим делением полученного результата на величину  $1/z$ .

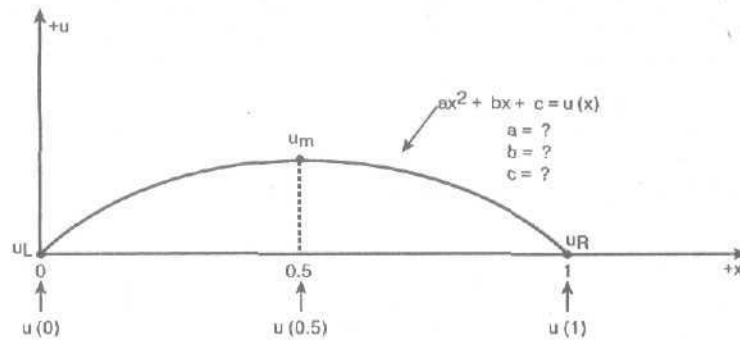


Рис. 12.30. Схема расчета кривой интерполяции

В результате мы получим, что точное значение координаты текстуры в средней точке можно записать следующим образом:

$$u_{mid} = \frac{u_l/z_l + u_r/z_r}{1/z_l + 1/z_r},$$

причем для вычислений можно использовать **линейную** интерполяцию. Для нас сейчас важно то, что мы легко можем найти значение  $u_{mid}$ , которое и дает нам третье условие для поиска кривой интерполяции, а именно то, что в средней точке интерполируемого отрезка интерполяция должна давать точное значение.

Итак, мы хотим найти коэффициенты  $a$ ,  $b$  и  $c$  функции  $f(x) = ax^2 + bx + c$  такие, что  $f(0) = u_l$ ,  $f(1) = u_r$  и  $f(0.5) = u_{mid}$ , так что мы получаем **следующую** систему уравнений:

$$\begin{aligned} c &= u_l, \\ a + b + c &= u_r, \\ a/4 + b/2 + c &= u_{mid}. \end{aligned}$$

Способ решения данной системы уравнения не важен; попробуйте решить ее самостоятельно. У вас должен получиться следующий ответ:

$$a = -4u_{\text{mid}} + 2u_l + 2u_r,$$

$$b = 4u_{\text{mid}} - 3u_l - u_r,$$

$$c = u_l.$$

Убедиться в правильности приведенного решения я предоставляю читателям в качестве самостоятельной работы. Аналогично, для координаты  $v$  получим:

$$a = -4v_{\text{mid}} + 2v_l + 2v_r,$$

$$b = 4v_{\text{mid}} - 3v_l - v_r,$$

$$c = v_l.$$

Итак, у нас есть все, что нам надо для выполнения интерполяции, — и при этом нет ни одного деления. Впрочем, два деления заменяются на четыре умножения и четыре сложения — поскольку мы должны вычислить для каждого пикселя в строке два значения:

$$u(x) = a_0x^2 + b_0x + c_0 = (a_0x + b_0)x + c_0,$$

$$v(x) = a_1x^2 + b_1x + c_1 = (a_1x + b_1)x + c_1.$$

## Вычисление правой разности для оценки квадратного многочлена

На практике для вычисления многочлена метод "в лоб" оказывается быстрее, чем два деления на каждый пиксель. Однако производительность можно улучшить с помощью других методов, например, такого как правая разность. При этом мы просто вычисляем  $u(0)$  и  $v(0)$ . Для вычисления правой разности нам нужно найти  $u(x+1)$ . Если известно значение  $u(x)$ , то мы можем предположить, что

$$u(x+1) = u(x) + du.$$

Итак, правая разность  $du$  вычисляется следующим образом:

$$du = u(x+1) - u(x),$$

или, с учетом квадратичности функции  $u(x)$ ,

$$\begin{aligned} du &= [a(x+1)^2 + b(x+1) + c] - [ax^2 + bx + c] = \\ &= 2ax + a + b. \end{aligned}$$

Таким образом,

$$u(x+1) = u(x) + (2ax + a + b).$$

Но это еще не все. Нам нужны два умножения (можно избежать одного умножения за счет сдвига вместо умножения на 2) и два сложения на один пиксель, итого — четыре умножения и четыре сложения на пиксель. Лучше не стало... Давайте продолжим и выполним еще одну итерацию — теперь уже для правой разности  $du$ :

$$du(x) = 2ax + a + b.$$

Мы хотим найти правую разность  $ddu$  такую, что  $du(x+1) = du(x) + ddu$ :

$$\begin{aligned} ddu &= du(x+1) - du(x) = \\ &= [2a(x+1) + a + b] - [2ax + a + b] = \\ &= 2a. \end{aligned}$$

Это постоянная величина, поэтому мы можем вычислять каждое новое значение  $u(x)$  следующим образом:

$$u(x+1) = u(x) + du(x),$$

$$du(x+1) = du(x) + ddu.$$

Итак, нам нужно просто найти  $u(0)$  и  $du(0)$ , и тогда мы сможем найти все правые разности и значения координат. При этом потребуется всего два сложения на пиксель для двух интерполянтов, т.е. всего четыре сложения на пиксель при интерполяции.

## Оптимизация текстурирования с помощью гибридных подходов

Теперь, когда у нас есть метод отображения текстуры с аффинной линейно-кусочной перспективой и точной перспективой, можно воспользоваться различными оптимизациями, которые состоят в вызове различных функций отображения текстуры в зависимости от типа отображаемой текстуры, а также от того, насколько близко объект расположен к точке наблюдения и какие искажения допустимы при визуализации. Например, для близкорасположенных объектов допустима только растеризация с точной перспективой, для объектов на средней дистанции — линейно-кусочная растеризация, а для удаленных объектов — аффинная (рис. 12.31).

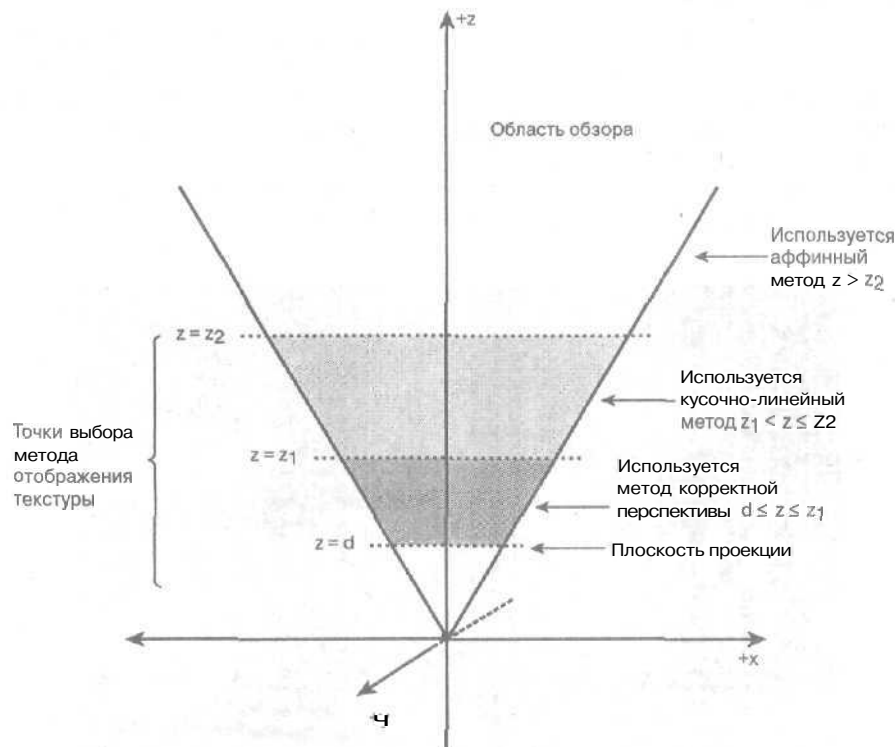


Рис. 12.31. Гибридное текстурирование на основании расстояния до объекта

Тем самым мы получим великолепную производительность и великолепную визуализацию. Реализация такой оптимизации тривиальна. При вызове функции списка визуализации (или без нее) определяется текущая координата  $z$  многоугольника (или ее сред-

нее значение). Это значение сравнивается с **соответствующими** значениями для различных методов отображения — точного, линейного и аффинного — и вызывается соответствующий растеризатор. Соответствующий псевдокод представлен ниже.

```
float perfect_range = 1000;
// 0-1000 единиц - идеальная текстура

float piecewise_range = 2000;
// 1000-2000 единиц - линейно-кусочный метод

float affine_range = 2000;
// 2000 и более единиц - аффинный метод

// выбор растеризатора
if (curr_poly->z < perfect_range)
    Perfect_Perspective_Texture_Mapper(curr_poly);
else if (curr_poly->z > perfect_range &&
        curr_poly->z < linear_range)
    Linear_Piecewise_Perspective_Texture_Mapper(curr_poly);
else if (curr_poly->z > affine_range)
    Affine_Perspective_Texture_Mapper(curr_poly);
```

Естественно, имена функций и доступ к информации о вершинах в реальном процессе отличается от приведенного псевдокода, но важен сам принцип. Это действительно хорошая оптимизация, и я реализовал ее в окончательном варианте высокоуровневой функции-оболочки. Однако я решил реализовать упрощенный вариант поддержки — только линейно-кусочного и аффинного методов.

В демонстрационной программе я вручную, "в лоб" реализовал описанный алгоритм. Имя демонстрационной программы — DEMOII12\_8.CPP|EXE, а копия ее экрана показана на рис. 12.32.

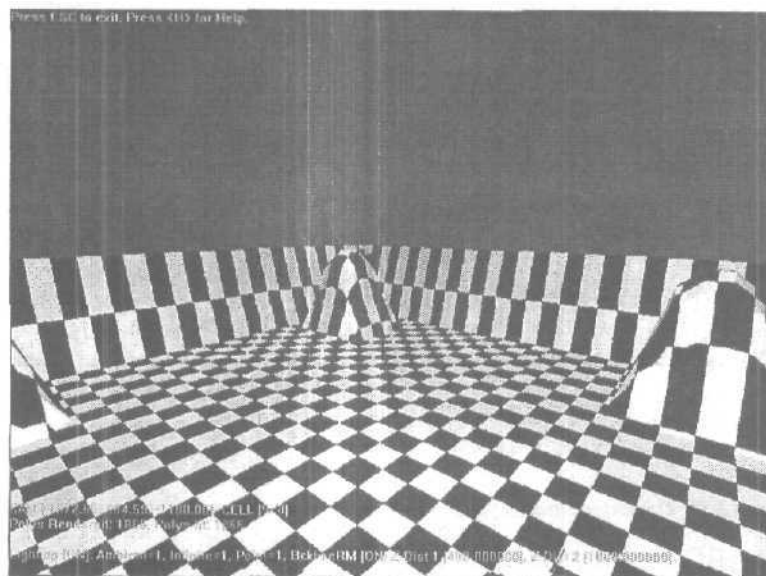


Рис. 12.32. Гибридный растеризатор текстуры

При работе программы можно менять расстояния, при которых вызываются различные растеризаторы, с помощью следующих клавиш:

- <1> — уменьшить первое расстояние;
- <2> — увеличить первое расстояние;
- <3> — уменьшить второе расстояние;
- <4> — увеличить второе расстояние.

Для компиляции программы нужны файлы DEMO112\_8.CPP\H, модули T3DLIB1-10.CPP\H и библиотечные файлы DirectX.

## Билинейная фильтрация текстуры

*Билинейная фильтрация текстуры* — это процесс выборки нескольких пикселей из исходной текстуры в процессе отображения с *последующим* усреднением или использованием некоторого другого фильтра для получения окончательного значения пикселя. В настоящее время при отображении текстуры мы используем выборку точек, т.е. мы вычисляем координаты текстуры  $u$  и  $v$ , отбрасываем дробную часть (или округляем ее) и затем используем их для доступа к текстелю в исходной текстуре (рис. 12.33).

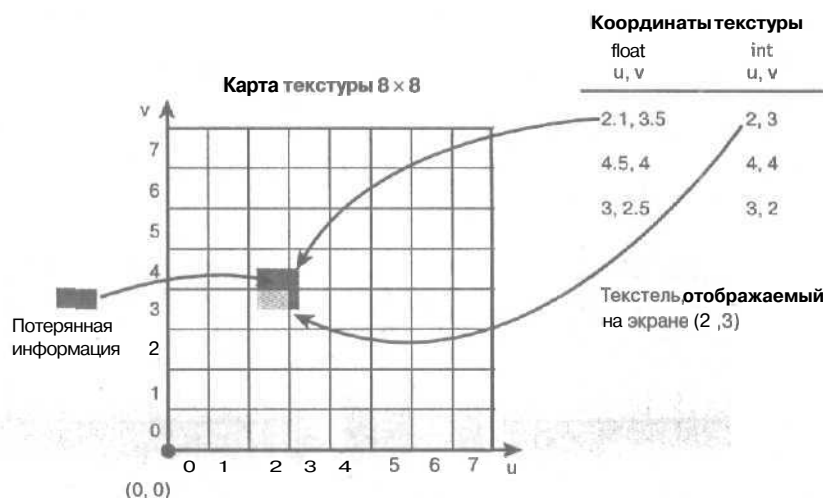


Рис. 12.33. Механизм точечной выборки

Проблема заключается в том, что часть информации, которую мы могли бы использовать для улучшения изображения, отбрасывается. В процессе же билинейной интерполяции или фильтрации вместо отбрасывания дробных частей координат текстуры  $u$  и  $v$  мы используем их для доступа к **текстелям** вокруг исходной точки в карте текстуры, а затем усредняем или линейно фильтруем их.

Например, предположим, что у нас есть карта текстуры 64x64 (рис. 12.34). Мы хотим сделать выборку четырех соседних **текстелей** и линейно усреднить их (или использовать, например, линейную интерполяцию с весовой функцией). На рис. 12.34 этот процесс показан визуально. Мы хотим вычислить конечный пиксель  $p_{\text{final}}$  по пикселям  $p_0$ ,  $p_1$ ,  $p_2$  и  $p_3$ , которые являются пикселями, окружающими интересующий нас исходный пиксель.

Мы решаем задачу, накладывая на исходную точку матрицу 2x2, составленную из пикселей  $p_0 - p_3$ . Затем мы определяем области перекрытия каждого из четырех пикселей и исходного и взвешенно суммируем их (само собой, в RGB-пространстве). На рис. 12.34 показано несколько примеров таких расчетов. Давайте рассмотрим их внимательно.

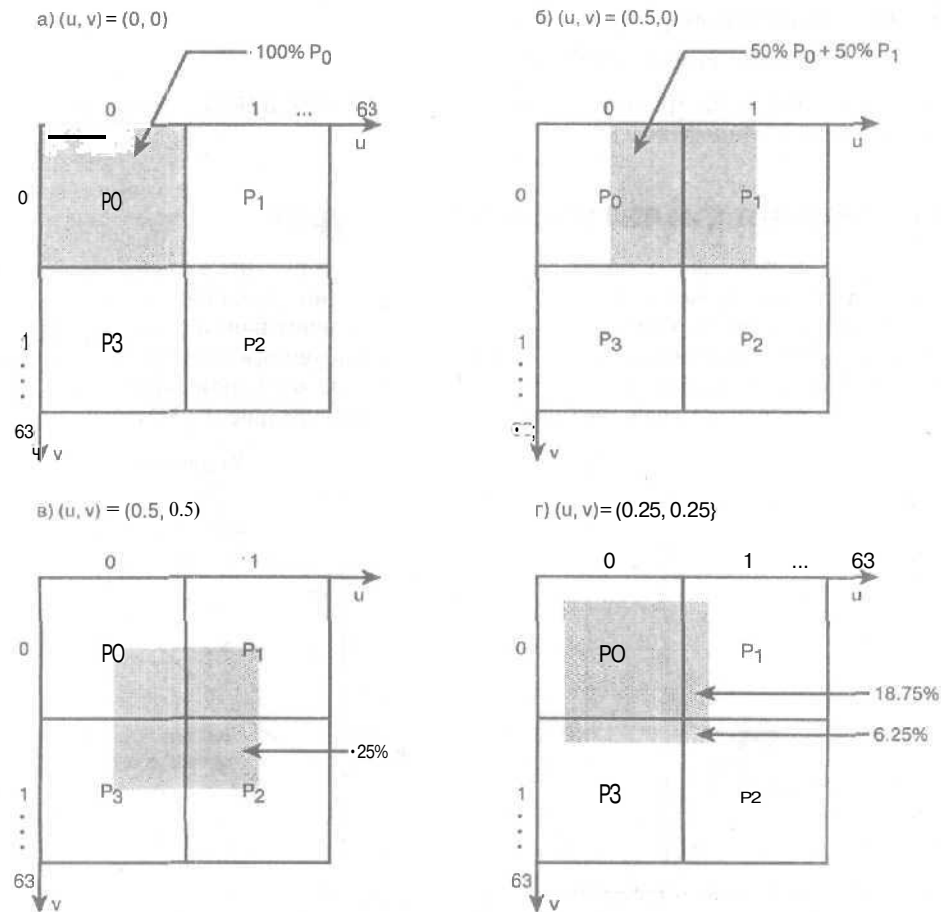


Рис. 12.34. Билинейная фильтрация

Без потери общности можно считать, что пиксельная матрица  $p_0 - p_3$  представляет собой верхний левый угол текстуры. Допустим теперь, что значения координат текстуры  $(u, v) = (0, 0)$ , что показано на рис. 12.34a. В этом случае блок выборки точно совпадает с текстелем  $(0, 0)$ , поэтому здесь взвешенное среднее равно

$$p_{\text{final}} = 1 \cdot p_0 + 0 \cdot p_1 + 0 \cdot p_2 + 0 \cdot p_3 = p_0.$$

Другой пример показан на рис. 12.34b. По сравнению с предыдущим примером, здесь произведен небольшой сдвиг по оси  $u$ , и  $(u, v) = (0.5, 0)$ . Следовательно, блок выборки перекрывает  $p_0$  и  $p_1$  на 50% каждый. Соответственно, окончательно получаем

$$p_{\text{final}} = 0.5 \cdot p_0 + 0.5 \cdot p_1 + 0 \cdot p_2 + 0 \cdot p_3 = 0.5 \cdot (p_0 + p_1).$$

На рис. 12.34а показан случай двумерной билинейной интерполяции:  $(u, v) = (0.5, 0.5)$ . Здесь блок выборки перекрывает 25% каждого **текстеля**, и

$$p_{\text{final}} = 0.25 \cdot p_0 + 0.25 \cdot p_1 + 0.25 \cdot p_2 + 0.25 \cdot p_3 = 0.25 \cdot (p_0 + p_1 + p_2 + p_3).$$

В этом случае мы должны учитывать не только перекрытие по оси  $u$ , но и по оси  $v$ , т.е. выполнить **билинейную** интерполяцию. Давайте попробуем **еще** раз (рис. 12.34б). Здесь  $(u, v) = (0.25, 0.25)$ . Рассматривая рисунок и подсчитывая **площади** областей **перекрытия**, находим

$$\begin{aligned} p_{\text{final}} &= (1 - 0.25)(1 - 0.25)p_0 + (1 - 0.25)(0.25)p_1 + \\ &\quad + (0.25)(0.25)p_2 + (0.25)(1 - 0.25)p_3 = \\ &= (0.75)^2 p_0 + (0.75 - 0.25)p_1 + (0.25)^2 p_2 + (0.25 - 0.75)p_3 = \\ &= 0.5625p_0 + 0.1875p_1 + 0.0625p_2 + 0.1875p_3. \end{aligned}$$

По сути, здесь одномерная линейная интерполяция  $(x) \cdot \text{value}_1 + (1 - x) \cdot \text{value}_2$ , применена нами к двум измерениям.

Нам нужна **общая** формула, которая позволит подставить любое значение  $u$  или  $v$  и билинейно интерполировать дробные компоненты  $u$  и  $v$ , выполнить выборку текстуры и даст нам окончательное значение пикселя. Ниже приводится псевдокод, который реализует ее для одиночного монохромного значения (**не** для полного RGB).

```
// Одноканальная билинейная интерполяция

// Монохромная текстура
float texture[TEXT_SIZE][TEXT_SIZE];

float u,v; // Координаты текстуры 0..TEXT_SIZE-1

// Вычисляем целые части значений  $u, v$ 
int ui = (int)u;
int vi = (int)v;

// Вычисляем дробные части значений  $u, v$ 
float du = u - ui;
float dv = v - vi;

// Сейчас мы готовы выполнить выборку текстуры
float pfinal = (1-du)*(1-dv)*texture[u][v]
              + (du)*(1-dv)*texture[u+1][v]
              + (du)*(dv)*texture[u+1][v+1]
              + (1-du)*(dv)*texture[u][v+1];
```

Как видно, проблема билинейной интерполяции состоит в большом количестве вычислений. Для одиночного монохромного канала нам нужно вначале **вычислить** целые и дробные части значений  $u$  и  $v$ , после чего найти дополнения дробных частей до единицы, и только после этого приступить к интерполяции, которая требует восьми умножений. Всего для каждого канала приходится шесть операций сложения и восемь операций умножения на пиксель. Нам нужны три канала, поэтому всего потребуется 18 сложений и 24 умножения на пиксель, что явно многовато... **Конечно**, с помощью специальных приемов оптимизации ее работу можно ускорить, однако даже при использовании таблиц поиска и т.п. остается слишком много интерполяций и извлечений данных.

Так что вопрос состоит в том, стоит ли реализовать билинейное отображение/фильтрацию текстур вообще? Этот механизм работает медленно, его можно использовать только в небольших количествах и не везде. Если нужно, чтобы хорошо выглядело только небольшое число объектов — это как раз тот случай, когда билинейность можно **применять**. 8 то же время похожего эффекта можно добиться путем небольшой размытости текстуры (blur). Конечно, это приводит к некоторому снижению детализации текстуры, так что хорошо подумайте, принимая то или иное решение.

Теперь я собираюсь реализовать билинейное **текстурирование** с помощью самых **простых растеризаторов** — аффинных с постоянным затенением без Z-буфера, с Z-буфером и с 1/z-буфером (всего три новых **растеризатора**). Бесполезно делать другие **растеризаторы** с поддержкой билинейной фильтрации, поскольку они и без того работают очень медленно. Я покажу только одну модификацию растеризатора текстуры (поскольку эти растеризаторы огромны). Если будет нужно, вы сами создадите другие версии, внося необходимые изменения в код.

Я думаю, **растеризатор** аффинной текстуры с постоянным затенением и Z-буфером будет наиболее **подходящим** объектом для наших модификаций. Поэтому я беру функцию `Draw_Textured_TriangleZB2_16()` и реализую ее версию с билинейной интерполяцией. Эта функция имеет самый простой внутренний цикл из всех текстурных растеризаторов. Прототип новой версии с поддержкой билинейной интерполяции приведен ниже.

```
void Draw_Textured_Bilerp_TriangleZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байт в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch)          // Байт в строке Z-буфера
```

Вызов функции остался прежним, просто к ее функциональности добавлена билинейная интерполяция текстуры. Я реализовал ранее показанный псевдокод алгоритма, с оптимизацией и в формате с фиксированной точкой. Ниже приведен листинг внутреннего цикла растеризатора, чтобы вы могли увидеть, как он работает.

```
// Выводим строку
for (xi=xstart; xi < xend; xi++)
{
    // Проверяем, не находится ли текущий пиксель ближе к
    // наблюдателю по координате z, чем находящийся в буфере
    if (zi < z_ptr[xi])
    {
        // Вычисляем целые части u,v
        int uint = ui >> FIXP16_SHIFT;
        int vint = vi >> FIXP16_SHIFT;

        int uint_pls_1 = uint+1;
        if (uint_pls_1 > texture_size)
            uint_pls_1 = texture_size;

        int vint_pls_1 = vint+1;
        if (vint_pls_1 > texture_size)
            vint_pls_1 = texture_size;

        int textel00 = textmap[(uint+0) +
                               ((vint+0) << texture_shift2)];
        int textel10 = textmap[(uint_pls_1) +
```

```

        ((vint+0) << texture_shift2]);
int texel01 = textmap[(uint+0) +
        ((vint_pls_1) << texture_shift2)];
int texel11 = textmap[(uint_pls_1) +
        ((vint_pls_1) << texture_shift2)];

// Извлекаем rgb-компоненты
int r_textel00 = ((texel00 >> 11));
int g_textel00 = ((texel00 >> 5) & 0x3f);
int b_textel00 = (texel00 & 0x1f);

int r_textel10 = ((texel10 >> 11));
int g_textel10 = ((texel10 >> 5) & 0x3f);
int b_textel10 = (texel10 & 0x1f);

int r_textel01 = ((texel01 >> 11));
int g_textel01 = ((texel01 >> 5) & 0x3f);
int b_textel01 = (texel01 & 0x1f);

int r_textel11 = ((texel11 >> 11));
int g_textel11 = ((texel11 >> 5) & 0x3f);
int b_textel11 = (texel11 & 0x1f);

// Вычисляем дробные части u,v в формате с фиксированной
// точкой 24.8
int dtu = (ui & (0xffff)) >> 8;
int dtv = (vi & (0xffff)) >> 8;
int one_minus_dtu = (1 << 8) - dtu;
int one_minus_dtv = (1 << 8) - dtv;

// У каждого интерполянта 3 члена - (du), (dv),
// текстур. Однако члены (du), (dv) повторяются при
// каждом цикле вычислений r_textel, g_textel и
// b_textel, так что мы можем вычислить их только один
// раз
int one_minus_dtu_x_one_minus_dtv = (one_minus_dtu) *
        (one_minus_dtv);
int dtu_x_one_minus_dtv = (dtu) * (one_minus_dtv);
int dtu_x_dtv = (dtu) * (dtv);
int one_minus_dtu_x_dtv = (one_minus_dtu) * (dtv);

// Теперь все готово для выполнения выборки текстуры
int r_textel = one_minus_dtu_x_one_minus_dtv*r_textel00+
        dtu_x_one_minus_dtv * r_textel10 +
        dtu_x_dtv * r_textel11 +
        one_minus_dtu_x_dtv * r_textel01;

int g_textel = one_minus_dtu_x_one_minus_dtv*g_textel00+
        dtu_x_one_minus_dtv * g_textel10 +
        dtu_x_dtv * g_textel11 +
        one_minus_dtu_x_dtv * g_textel01;

int b_textel = one_minus_dtu_x_one_minus_dtv*b_textel00+
        dtu_x_one_minus_dtv * b_textel01 +
        dtu_x_dtv * b_textel11 +
        one_minus_dtu_x_dtv * b_textel01;

```

```

// Записываем текстель
screen_ptr[xi] - ((r_textel >> 16) << 11) +
                ((g_textel >> 16) << 5) +
                (b_textel >> 16);

// Обновляем Z-буфер
z_ptr[xi] = zi;
} // if

```

Код вычисляет доли перекрытия и производит усреднение в **RGB-пространстве**; результаты расчета выводятся на экран. Проблема алгоритма не в **сложности**, а в большом **объеме** работы, в результате чего его трудно оптимизировать. По сути, мы выполняем алгоритм обработки изображения, а такие алгоритмы в реальном времени никогда хорошо не работают. В коде есть ряд оптимизаций, таких как предварительное вычисление **общих коэффициентов** и использование формата с фиксированной точкой 8.24. Этот новый формат необходим, поскольку во время умножения каждого билинейного **члена** есть три произведения, два из них — в формате с фиксированной точкой **16.16**, что приводит к переполнению. Поэтому мне пришлось немного его сдвинуть и работать с форматом 8.24. Тем не менее, результаты получились весьма **впечатляющими**, и соответствующая демонстрационная программа выглядит просто замечательно!

Кроме того, ниже приводятся прототипы двух других модифицированных функций с добавленной поддержкой билинейной интерполяции: версия с инверсным **Z-буфером** и версия без сортировки (обе для аффинных текстур и постоянным затенением).

```

void Draw_Textured_Bilerp_TriangleINVZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,      // Байтов в строке, 320, 640 и т.д.
    UCHAR *zbuffer,     // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

```

```

void Draw_Textured_Bilerp_Triangle_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch);     // Байтов в строке, 320, 640 и т.д.

```

Демонстрационная программа билинейной **интерполяции** — **DEMOII12\_9.CPP|EXE** (рис. 12.35). Программа позволяет переключаться между режимами точечной выборки и билинейной фильтрации текстуры объекта с помощью клавиши <B>. Для выбора объекта можно использовать клавишу <N>. Интересно, увидите ли вы отличие на экране? Для компиляции программы необходимы файлы **DEMOII12\_9.CPP|Н**, **T3DLIB1-10.CPP|Н**, а также библиотечные файлы **DirectX**.

## Множественное отображение и трилинейная фильтрация текстур

Название множественного отображения в оригинале — **mip mapping** — основано на латинском изречении “**multum in parvo**”, что означает “много вещей в малом объеме”. Проблема, которую решает множественное отображение, заключается в том, что при отображении текстуры на многоугольник и выборке текстуры (с помощью точечной выборки, билинейной фильтрации и т.д.) возникает ряд побочных эффектов, связанных с

выборкой пространственной информации. Одна из таких проблем — "сверкание" текстуры, которое возникает, когда в поле зрения попадают высокочастотные элементы текстуры (т.е. происходят большие изменения интенсивности при изменении координаты точки), что обычно происходит при уменьшении масштаба.

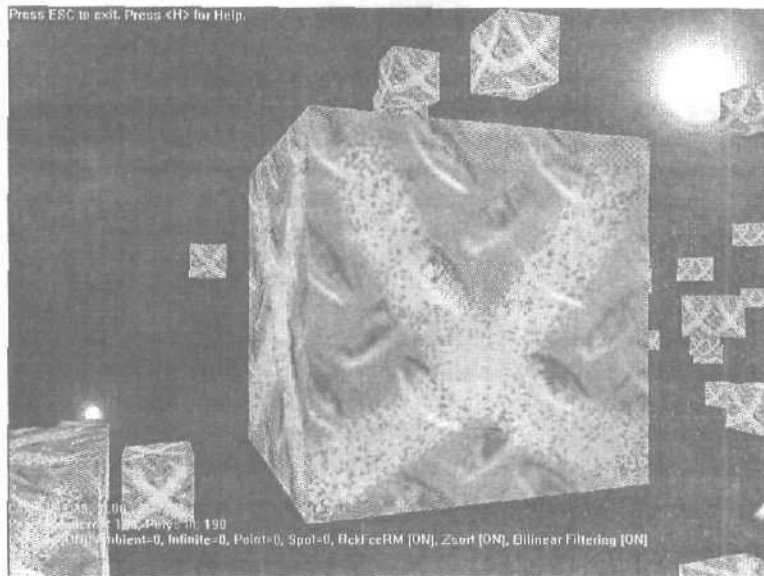


Рис. 12.35. Билинейная интерполяция устраняет "неровности"

Второй серьезной проблемой отображения текстуры является низкочастотное наложение, приводящее к появлению муара. Это происходит, когда низкочастотное изображение формируется на основе высокочастотных оригиналов. На рис. 12.36 показана шахматная текстура с точечной выборкой. Вы заметили наличие на рисунке синусоидального узора? Теперь посмотрите на рис. 12.37 — на нем показана та же сцена, но со множественным отображением и фильтрацией текстуры. Эффект гораздо менее выражен, а изображение значительно чище.

#### НА ЗАМЕТКУ

Когда два повторяющихся шаблона линий, кругов или массивов точек накладываются с неточным совмещением, возникает узор из светлых и темных линий, называемый муаром. Муар — это не изображение, а скорее оптическая иллюзия. Обычно узор возникает в местах высокого контраста или высокой частоты компонентов. Муар возникает также при наложении двух изображений вследствие интерференции светлых и темных областей.

Для реализации множественного отображения мы создадим то, что называется цепочкой множественного отображения текстур (mip chain), в которой каждая последующая текстура в два раза меньше предыдущей по каждой оси. Размер последней текстуры — 1x1. Кроме того, мы генерируем каждую из этих текстур с помощью фильтра (усреднения, Гаусса и т.п.). Затем на основе некоторой метрики мы выбираем подходящую текстуру — так же, как визуализация многоугольников у нас зависит от расстояния до них от точки обзора или площади проекции многоугольника (подробнее об этом будет сказано позже). Таким образом минимизируется как сверкание, так и низкочастотный муар. Так выглядит наш план в общих чертах; а теперь рассмотрим его подробнее.

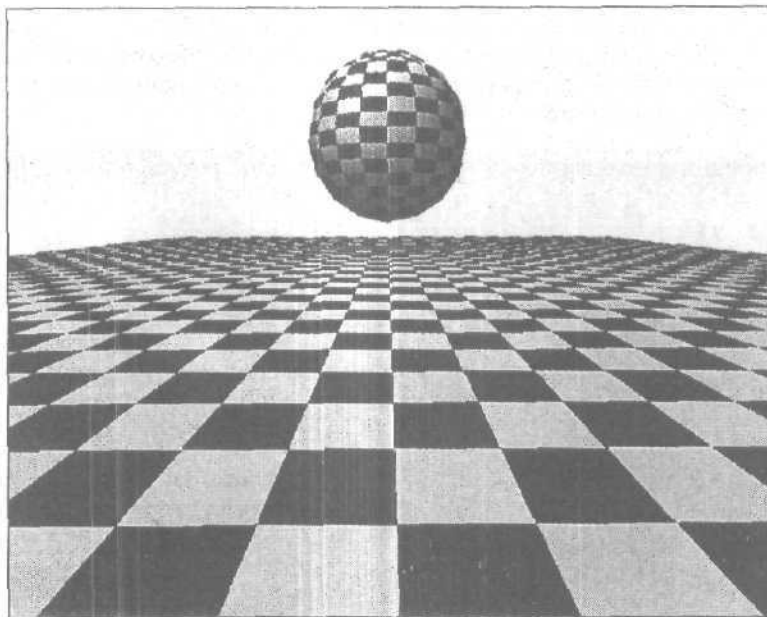


Рис. 12.36. Изображения с точечной выборкой выглядят не лучшим образом

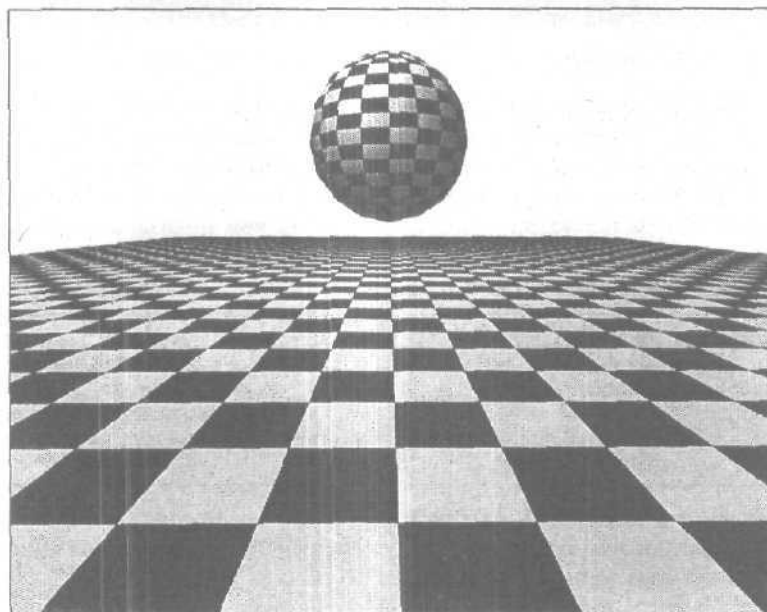


Рис. 12.37. Изображения со множественным отображением с фильтрацией выглядят гораздо лучше

## Введение в Фурье-анализ

Вначале я планировал подробно **объяснить** Фурье-анализ, однако после множества лекций по математике и электротехнике, прочитанных в колледже, я пришел к выводу, что вряд ли удастся сделать это менее чем на 100 страницах.

Так что вместо этого я дам вам некоторую информацию о том, почему это **важно**, что при этом происходит, о чем это говорит и почему мы этим интересуемся.

### НА ЗАМЕТКУ

Если вас действительно интересует **дополнительная информация по Фурье-анализу**, я настоятельно рекомендую следующую **литературу**: Mischa Schwartz. **Information, Transmission, Modulation and Noise**. McGraw Hill, а также McGillem and Cooper. **Continuous and Discrete Signal and System Analysis**. Oxford University Press. Дополнительные **объяснения с точки зрения компьютерной графики** содержатся в книгах Foley et al. **Computer Graphics: Principles and**. Addison-Wesley, а также George Wolberg, **Digital Image Warping**. Wiley-IEEE Press.

Фурье-анализ в целом **посвящен** разложению сигналов во временной области (или пространстве, если речь идет о графике) на составляющие синусоидальные компоненты. При этом любой периодический сигнал в большинстве случаев (при выполнении условий Дирихле) может быть разложен на синусоидальные волны различной частоты, амплитуды и фазы. Иными словами, любой сигнал является суммой синусоидальных волн, и поэтому мы можем записать любой сигнал в виде ряда Фурье, который представляет собой сумму синусоидальных членов. Итак, любой сигнал можно представить рядом Фурье. Да, я знаю, что хожу вокруг да около, но что делать?

Преобразование Фурье позволяет нам извлечь из сигнала необходимую информацию и вычислить составляющие его части. Допустим, у нас есть чисто синусоидальная волна с частотой  $f$  и периодом  $T = 1/f$  (рис. 12.38a).

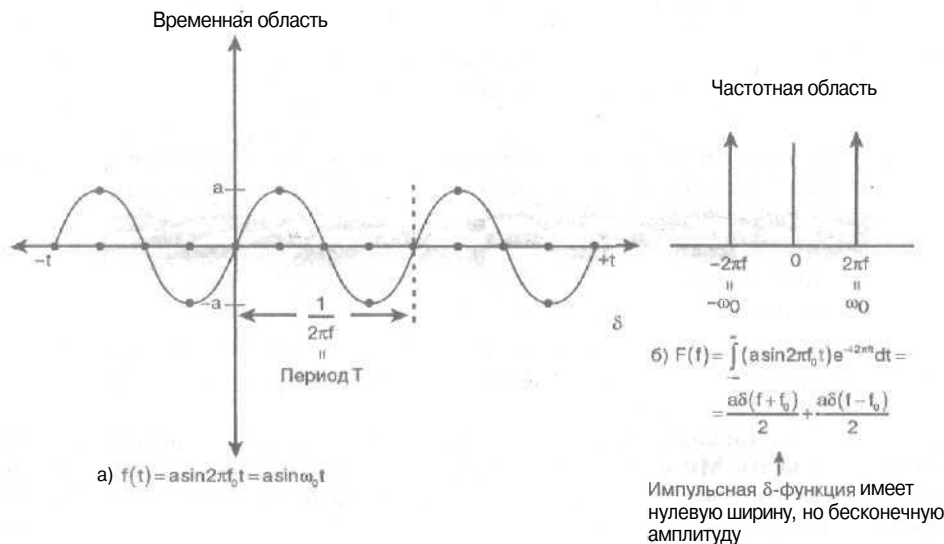


Рис. 12.38. Синусоида и ее Фурье-преобразование

Этот сигнал находится во временной области, т.е. у нас есть функция от **времени**, которая выглядит следующим образом ( $t$  — время,  $T$  — период);

$$f(t) = a \sin(\omega t) = a \sin(2\pi f t) = a \sin \frac{2\pi}{1} t.$$

Здесь  $t$  — время,  $a$  — амплитуда сигнала, а  $f$  — его частота. Если выполнить преобразование Фурье этой простой синусоиды, получится одиночное значение (плюс его мнимое сопряженное), расположенное на частоте исходной синусоидальной волны, с амплитудой  $a$ . Таким образом, Фурье-образ изображается не на оси времени, а на оси частоты. Рис. 12.38б показывает Фурье-образ на оси частоты для синусоиды. Если выполнить преобразование Фурье для функции

$$f(t) = 10\sin(100t) + 20\sin(200t),$$

то мы получим два импульса, расположенных на оси частоты на отметках 100 и 200, и соответственно -100 и -200. Любые синусоидальные волны дают похожий результат, а любой сигнал может быть разбит на синусоидальные волны, а затем вновь реконструирован в оригинал. Математическая операция, выполняющая это преобразование сигнала из временной области в частотную, имеет следующий вид:

$$F\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{-2\pi i f t} dt.$$

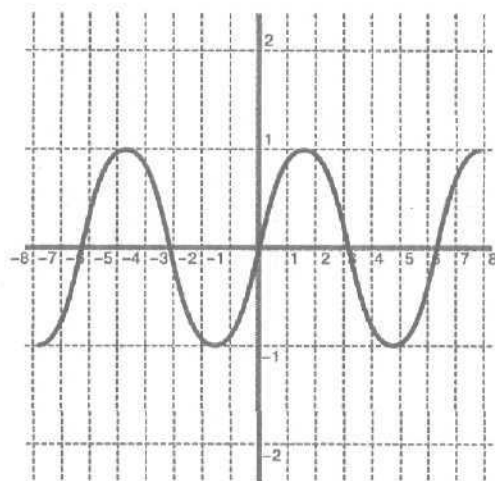
Для поиска преобразования Фурье нужно просто взять функцию  $f(t)$  и проинтегрировать ее в соответствии с приведенной формулой. В общем, чтобы выполнить множественное отображение, вы должны уметь выполнить преобразование Фурье, суть которого состоит в извлечении частотных характеристик сигнала. Всю эту работу уже сделали за нас (какое счастье!), так что поговорим только о результатах.

#### СОВЕТ

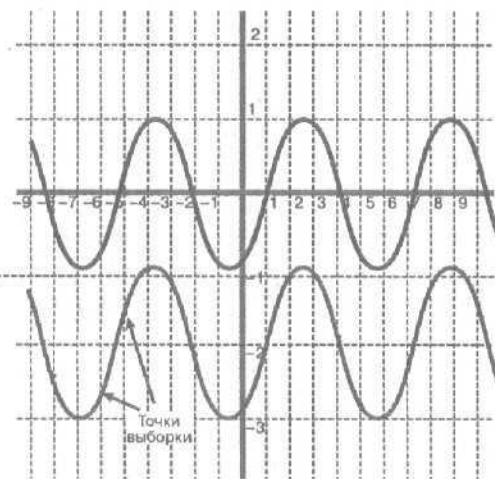
Графики на рис. 12.39 были выполнены с помощью MathGV, свободно распространяемой графической программы, такой же мощной, как и профессиональные пакеты. Попробуйте работать с ней — она есть на компакт-диске в каталоге Applications\ (ее можно также найти по адресу <http://www.mathgv.com>).

Посмотрите на рис. 12.39. На рис. 12.39а показан сигнал, из которого производится выборка (это может быть звуковой сигнал, видеосигнал и т.п.). Сигнал имеет частоту  $f_0$  и период  $T_0 = 1/f_0$ . На рис. 12.39б мы делаем выборку сигнала с некоторой частотой  $f_1 > 2 \cdot f_0$ . Как видно из рис. 12.39б, при этом по выборке можно без проблем реконструировать исходный сигнал. Если мы понизим частоту выборки до  $2 \cdot f_0$  (рис. 12.39в), то можно получить ситуацию, когда реконструированный сигнал окажется просто прямой линией! Посмотрите теперь на рис. 12.39г. Здесь выборка производится с частотой  $f_1 < 2 \cdot f_0$ , и реконструированная волна может вполне оказаться не исходным, а некоторым ложным сигналом, частота которого меньше, чем у исходного. По сути, сейчас я проиллюстрировал вам теорему Котельникова о том, что для восстановления сигнала с частотой, не превышающей  $f_0$ , выборка должна осуществляться с частотой, превышающей удвоенную частоту  $f_0$ .

Проблема в том, что обойти эту теорему невозможно и определенный ложный сигнал всегда будет возникать. Мы можем только минимизировать его, для чего следует удалить высокочастотные компоненты, которые вызывают ложный сигнал при минимизации текстуры. Именно это и есть основой множественного наложения: создать цепочку текстур, основанных на исходной текстуре, с последовательно уменьшающейся частотой выборки информации. При этом каждая последующая текстура имеет размер в четыре раза меньший, чем предыдущая.



а) Исходный сигнал



б) Выборка с частотой  $f_1 > 2 \cdot f_0$

Рис. 12.39. Выборка сигнала с разной частотой

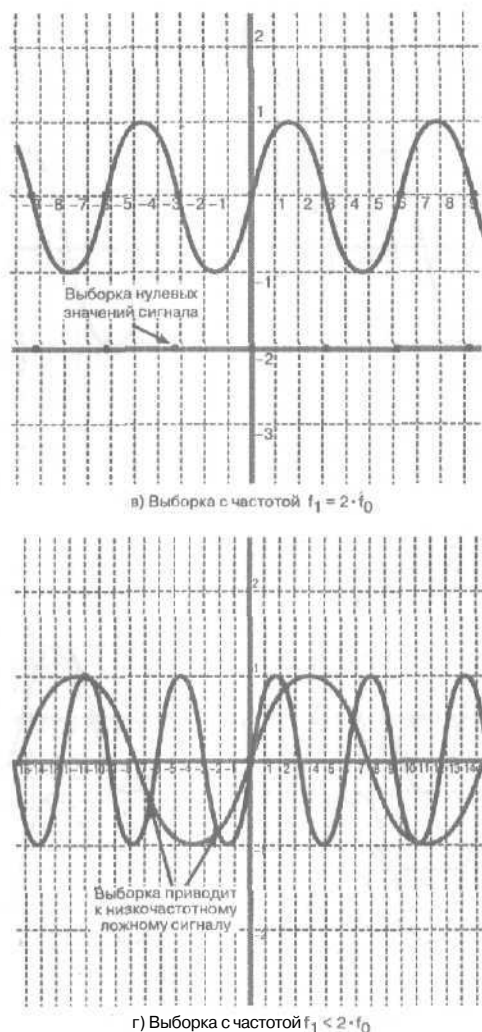


Рис. 12.39. Выборка сигнала с разной частотой (окончание)

## Создание цепочки множественного отображения

Итак, мы берем базовую текстуру (назовем ее  $T_0$ ) и создаем на ее основе цепочку текстур  $T_1, T_2, \dots, T_n$  (обычно путем усреднения), в которой каждая последующая текстура имеет размер в четыре раза меньше предыдущей (т.е. для создания текстуры  $T_{k+1}$  на основе текстуры  $T_n$  мы делим ширину и высоту текстуры  $T_n$  на 2, как показано на рис. 12.40).

Для вычисления значения пикселей новой текстуры на основе текстуры предыдущего уровня используем фильтр усреднения, т.е. берем пиксели с координатами  $(x, y)$ ,  $(x+1, y)$ ,  $(x, y+1)$  и  $(x+1, y+1)$  и усредняем их в RGB-пространстве. Полученное значение используется в качестве пикселя новой текстуры. Эта операция показана на рис. 12.40.

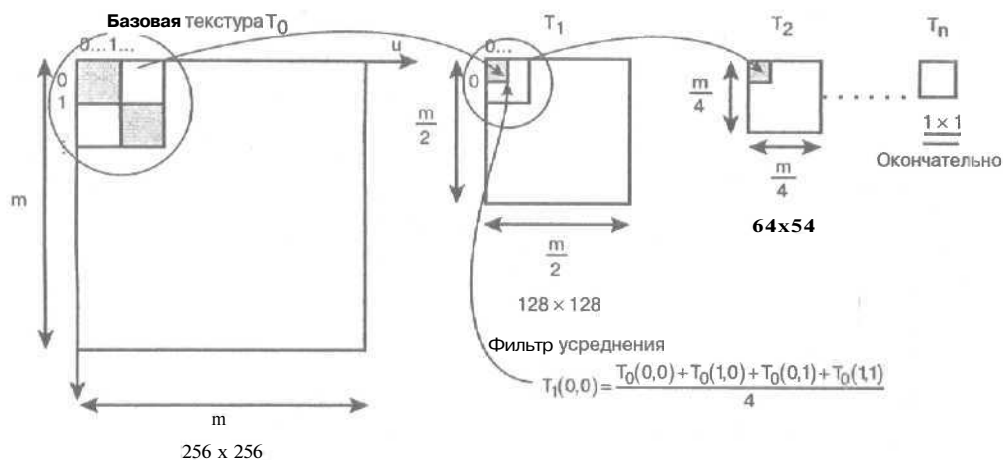


Рис. 12.40. Создание цепочки текстур множественного отображения

Приведу несколько правил создания цепочек множественного отображения. Во-первых, все текстуры должны быть квадратными и иметь линейные размеры, равные степени 2. Благодаря этому уменьшение размеров текстуры будет всегда одинаково, и конечная текстура всегда имеет размер  $1 \times 1$ . По соглашению исходную текстуру мы называем текстурой нулевого уровня, а из нее последовательно порождаются текстуры более высокого уровня — 1, 2, 3, 4, ...,  $n$ . Например, пусть у нас есть текстура размером  $256 \times 256$ . Размеры остальных текстур в цепочке показаны в табл. 12.4.

Таблица 12.4. Размеры текстур в цепочке с базовым размером  $256 \times 256$

Уровень	Размер текстуры
0	$256 \times 256$
1	$128 \times 128$
2	$64 \times 64$
3	$32 \times 32$
4	$16 \times 16$
5	$8 \times 8$
6	$4 \times 4$
7	$2 \times 2$
8	$1 \times 1$

Общее число уровней, включая исходную текстуру, можно вычислить с помощью уравнения 12.4.

Уравнение 12.4. Общее число уровней отображения, включая базовую текстуру

$$\log_2 T_0$$

В нашем случае  $\log_2 256 = 8$ . Чтобы найти общее число текстур, нужно просто добавить 1 к числу, получаемому по формуле 12.4. Итак, общее число текстур равно  $(\log_2 T_0 + 1)$ ; в нашем случае — 9.

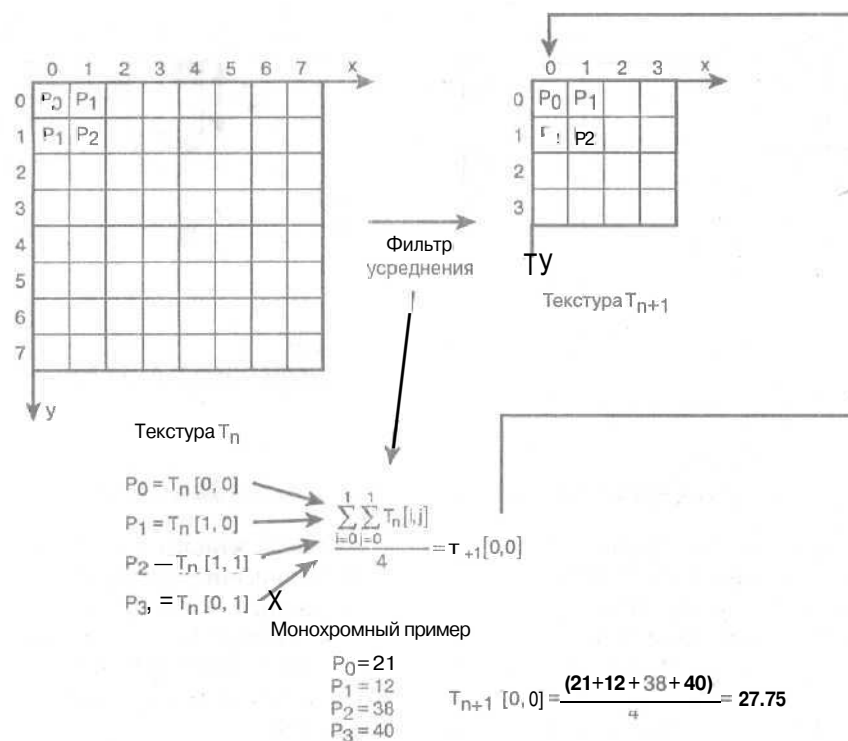


Рис. 12.41. Использование усреднения для создания множественного отображения

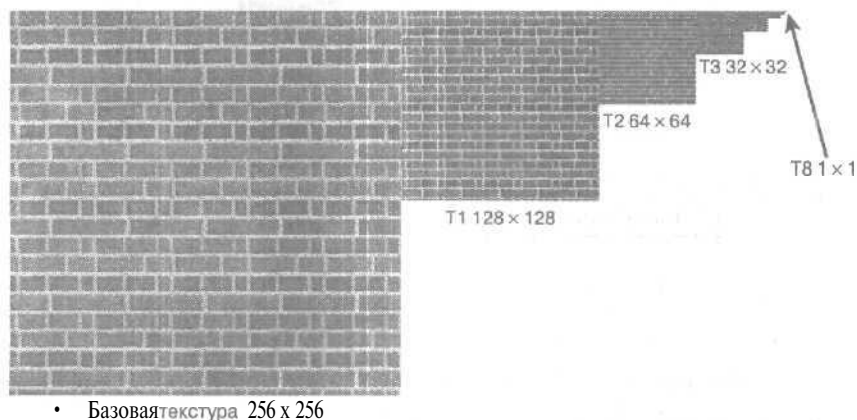


Рис. 12.42. Пример множественного отображения

А что можно сказать об использовании памяти? Посмотрим на рисунок 12.42, на котором показан пример множественного отображения. Как видите, размер каждой последующей текстуры равен 1/4 размера предыдущей. Таким образом, дополнительный расход памяти незначителен. Просчитаем дополнительный расход памяти для рассмотренного случая.

Если размер исходной текстуры  $256 \times 256$  ( $256^2$  слов), объем памяти, необходимый для хранения последовательности текстур множественного отображения, в которой каждая последующая текстура в 4 раза меньше предыдущей, вычисляется следующим образом:

$$x = 256^2 + 128^2 + 64^2 + 32^2 + 16^2 + 8^2 + 4^2 + 2^2 + 1^2,$$

Память, необходимая для хранения исходной текстуры, —  $256^2$ . Получается, что памяти нам требуется примерно в 1.333 раз больше:

$$\frac{x}{256^2} \sim \frac{256^2 + 128^2 + 64^2 + 32^2 + 16^2 + 8^2 + 4^2 + 2^2 + 1^2}{256^2} \approx 1.333$$

Иными словами, добавление текстур множественного отображения увеличивает расход памяти на 33%. Небольшая плата за те возможности, которые мы получаем.

Теперь, когда мы знаем, как создавать множественное отображение и сколько памяти для этого нужно, необходимо добавить эти возможности к игровому процессору. Я много думал об этом и пришел к выводу, что нужно снова менять структуры **объектов/многоугольников**. Но потом мне пришла в голову одна идея. Мы можем установить указатель текстуры не на одиночную текстуру, а на цепочку множественного отображения. Затем в каждом многоугольнике, участвующем во множественном отображении, мы устанавливаем соответствующий флаг и, когда выполняется визуализация многоугольника, вместо использования указателя на текстуру как на растровое изображение, мы используем его как указатель на массив указателей на растровые изображения! Так мы используем те же **структуры**, что и раньше, и ничего не теряем. Посмотрите на рис. 12.43, чтобы понять, о чем я говорю.

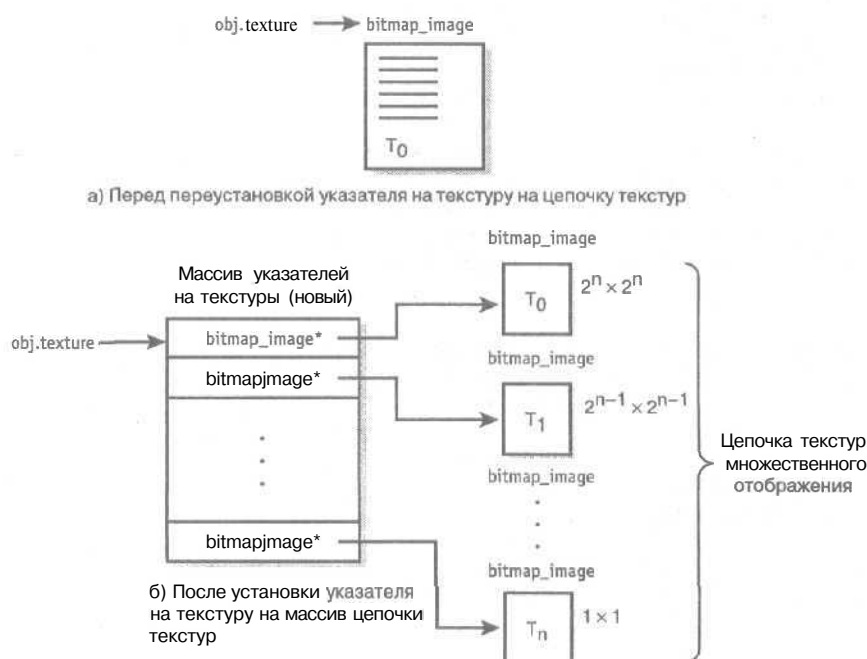


Рис. 12.43. Массив указателей на текстуры, поддерживающий цепочку множественного отображения

Когда текстура участвует во множественном отображении, указатель на первоначальную текстуру устанавливается на текстуру уровня 0. Затем, в два этапа, создается цепочка

множественного отображения. Вначале нужно выделить массив указателей на растровые изображения, затем установить указатель на исходную текстуру на этот массив. После этого мы должны выделить память для каждой текстуры, сгенерировать ее и установить каждый указатель в массиве множественного отображения на растровое изображение соответствующего уровня. Выглядит довольно запутанно, но это не так.

Единственная проблема во всей этой схеме состоит в том, что мы заставляем указатель на растровое изображение указывать не на растровое изображение, а на массив указателей на растровые изображения. Таким образом, нам нужно быть предельно внимательными и следить за тем, чтобы функции корректно работали с указателями, в частности, функции, которые не поддерживают множественное отображение, должны уметь использовать в качестве текстуры уровень 0 передаваемой **цепочки**.

В реализации описанного метода есть некоторые тонкости; о них вы вскоре узнаете. А пока рассмотрим код функции генерации цепочки множественного отображения.

```
int Generate_Mipmaps(
    BITMAP_IMAGE_PTR source,          // Исходное растровое изображение
    BITMAP_IMAGE_PTR *mipmaps,        // Указатель на массив цепочки
                                       // множественного отображения
    float gamma)                       // Коэффициент гамма-коррекции
{
    // Эта функция создает цепочку множественного
    // отображения растровых текстур. Функция получает
    // указатель на текстуру уровня d - 0. При выходе из
    // функции mipmaps указывает на массив указателей,
    // который содержит все уровни, включая исходный уровень
    // 0. Кроме того, функция возвращает общее число
    // уровней, или -1 в случае возникновения проблем.
    // Последний параметр gamma используется для
    // подсвечивания каждого уровня отображения, поскольку
    // усреднение дает эффект затемнения. Значения 1.01
    // обычно достаточно, значения более 1.0 увеличивают
    // яркость уровней множественного отображения, значения
    // менее 1.0 затемняют их, значение 1.0 не дает никакого
    // эффекта

    BITMAP_IMAGE_PTR *tmipmaps;        // Временный локальный
                                       // указатель на массив
                                       // указателей

    // Шаг 1: вычисляем общее количество уровней
    // множественного отображения
    int num_mip_levels = logbase2ofx[source->width] + 1;

    // Выделение памяти для массива указателей
    tmipmaps = (BITMAP_IMAGE_PTR *)malloc(num_mip_levels *
        sizeof(BITMAP_IMAGE_PTR));

    // Устанавливаем элемент 0 на исходный объект
    tmipmaps[0] = source;

    // Устанавливаем ширину и высоту (они одинаковы)
    int mip_width = source->width;
```

```

int mip_height = source->height;

// Выполняем итерацию и генерируем уровни множественного
// отображения с помощью фильтра усреднения
for (int mip_level = 1; mip_level < num_mip_levels;
    mip_level++)
{
    // Уменьшаем размер текстуры на один уровень
    mip_width = mip_width / 2;
    mip_height = mip_height / 2;

    // Выделяем память для растрового объекта
    tmipmaps[mip_level] =
        (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));

    // Создаем растровое изображение
    Create_Bitmap(tmipmaps[mip_level], 0, 0, mip_width,
        mip_height, 16);
    SET_BIT(tmipmaps[mip_level]->attr,
        BITMAP_ATTR_LOADED);

    // Выполняем усреднение пикселей предыдущего уровня
    // для создания данного
    for (int x = 0; x < tmipmaps[mip_level]->width; x++)
    {
        for (int y = 0; y < tmipmaps[mip_level]->height;
            y++)
        {
            // Мы должны усреднить пиксели предыдущего
            // уровня
            // (x*2, y*2), (x*2+1, y*2),
            // (x*2, y*2+1), (x*2+1, y*2+1)
            // и внести их в изображение текущего
            // уровня. Это так просто!

            float r0, g0, b0, // rgb-компоненты
                r1, g1, b1, // 4 пикселей выборки
                r2, d2, b2,
                r3, g3, b3;

            int r_avg, g_avg, b_avg; // Средние значения

            USHORT *src_buffer =
                (USHORT *)tmipmaps[mip_level-1]->buffer,
                *dest_buffer =
                (USHORT *)tmipmaps[mip_level]->buffer;

            // Извлечение rgb-компонентов каждого пикселя
            _RGB565FROM16BIT(src_buffer[(x*2+0)+
                (y*2+0)*mip_width*2],
                &r0, &g0, &b0);
            _RGB565FROM16BIT(src_buffer[(x*2+1)+
                (y*2+0)*mip_width*2],

```

```

        &r1, &g1, &b1);
    _RGB565FROM16BIT(src_buffer[(x*2+0)+
        (y*2+1)*mip_width*2],
        &r2, &g2, &b2);
    _RGB565FROM16BIT(src_buffer[(x*2+1)+
        (y*2+1)*mip_width*2],
        &r3, &g3, &b3);

    // Вычисляем средние значения с учетом gamma
    r_avg = (int)(0.5f + gamma*(r0+r1+r2+r3)/4);
    g_avg = (int)(0.5f + gamma*(g0+g1+g2+g3)/4);
    b_avg = (int)(0.5f + gamma*(b0+b1+b2+b3)/4);

    // Ограничиваем rgb-значения
    if (r_avg > 31) r_avg = 31;
    if (g_avg > 31) g_avg = 31;
    if (b_avg > 31) b_avg = 31;

    // Записываем данные
    dest_buffer[x+y*mip_width] =
        _RGB16BIT565(r_avg,g_avg,b_avg);
    } // for y
} // for x
} // for mip_level

// Присваиваем массив выходных указателей
*mipmaps = (BITMAP_IMAGE_PTR)tmipmaps;

// Возвращаем код успешного завершения
return(num_mip_levels);
} // Generate_Mipmaps

```

Функция получает три параметра. Первый из них — это указатель на исходное растровое изображение. Это может быть указатель на любой реальный объект типа `BITMAP_IMAGE`. Однако не забывайте, что это всего лишь указатель! Второй параметр немного более сложный:

```
BITMAP_IMAGE_PTR *mipmaps;
```

Это — указатель на указатель на `BITMAP_IMAGE`. Позвольте мне пояснить. Предположим, у нас есть объект (в данном случае **многоугольник**), у которого есть указатель на текстуру. Указатель указывает на `BITMAP_IMAGE`, т.е. сам структурный элемент в объекте или многоугольнике — это `BITMAP_IMAGE_PTR` (иными словами, `BITMAP_IMAGE*`). Проблема в том, что когда функция генерирует все объекты множественного отображения, нам нужно поместить в указатель адрес цепочки множественного отображения. Но так как параметры передаются в функции по значению, для того, чтобы изменить значение некоторого указателя, нужно передать в функцию его адрес. Таким образом, нужно передать указатель типа `BITMAP_IMAGE**`.

Я сделал функцию достаточно логичной и гибкой, так что один и тот же объект можно использовать и в качестве исходной текстуры, и для цепочки множественного отображения. Надо лишь корректно передать — в первом случае значение указателя, во втором — его адрес, например, следующим образом.

```
Generate_Mipmaps(obj->texture,
  (BITMAP_IMAGE_PTR *)&obj->texture,
  1.01);
```

Здесь `obj` — это объект типа `OBJECT4DV2_PTR`, в котором есть поле, являющееся указателем на `BITMAP_IMAGE` (т.е. имеющее тип `BITMAP_IMAGE_PTR`). Мы передаем значение указателя в качестве первого параметра, и его адрес — в качестве второго. Потенциально это может привести к потере исходных данных, на которые указывал этот указатель. Чтобы этого избежать, мы будем использовать исходное изображение в качестве первой текстуры цепочки. При удалении цепочки мы вернем указатель на это изображение указателю `obj->texture`, так что никаких потерь информации не будет. Схема описанной процедуры показана на рис. 12.44.

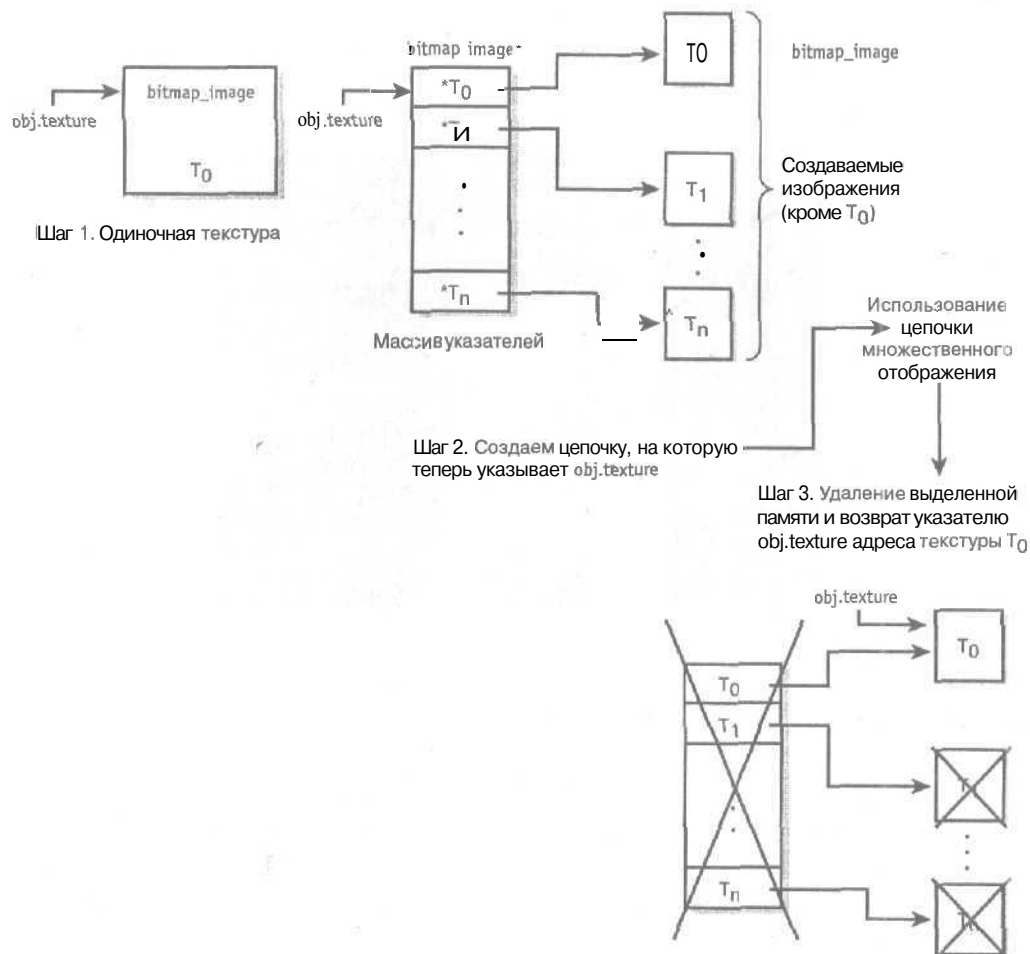


Рис. 12.44. Модификация указателя на текстуру объекта

Вернемся к функции. Она создает цепочку множественного отображения, выделяя память для массива указателей на текстуры, а затем итеративно вычисляя размер очередного уровня множественного отображения, выделяя для него память и выполняя усреднение в

RGB-пространстве. Этот процесс повторяется, пока не будет достигнут размер текстуры 1x1, после чего функция завершает работу. Последний параметр функции (который имеет значение по умолчанию, равное 1.01) — гамма-уровень. Дело в том, что, когда происходит усреднение изображения, оно становится менее ярким, и гамма-уровень обычно используется для повышения яркости каждого последующего уровня для компенсации ее снижения из-за усреднения. Чтобы продемонстрировать этот эффект, я бы хотел немного познакомить вас с демонстрационной программой DEMOII12\_10.CPP|EXE, которая позволяет выбирать различные текстуры, на основе которых она "на лету" создает цепочку множественного отображения и отображает ее на экране (рис. 12.45). Кроме того, она позволяет изменить уровень гамма-коррекции и тут же увидеть результат на экране. Клавиши управления курсором в программе имеют следующие значения:

- вправо — выбрать следующую текстуру;
- влево — выбрать предыдущую текстуру;
- вверх — повысить гамма-уровень;
- вниз — понизить гамма-уровень.

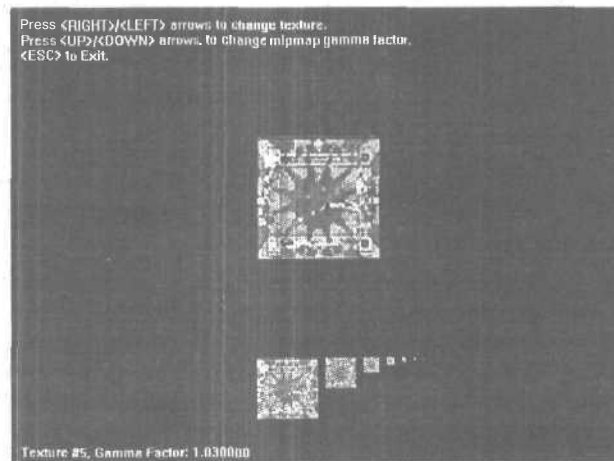


Рис. 12.45. Копия экрана генератора множественного отображения

#### СОВЕТ

Каждый уровень множественного отображения вычисляется путем усреднения предыдущей текстуры от размера  $(m \times m)$  до размера  $(m/2 \times m/2)$ . Это приводит к небольшому ослаблению интенсивности изображения. Чтобы нейтрализовать этот эффект, используется уровень гамма-коррекции, работающий как усилитель интенсивности во время процесса усредняющей фильтрации. Благодаря этому *текстелы* немного увеличивают свою яркость.

Для компиляции программы необходимы файлы DEMOII12\_10.CPP|H, T3DLIB1-10.CPP|H и библиотечные файлы DirectX.

Вернемся к нашим пояснениям. Итак, `obj->texture` после вызова `Generate_Mipmaps()` указывает на цепочку множественного отображения, а не на одиночную текстуру. Таким образом, если просто вызвать растеризатор, ничего хорошего не получится. Нужно каким-то образом пометить многоугольники, участвующие в множественном отображении, и учитывать эту особенность при вызове растеризатора. Для этого я ввожу два новых флага — один для многоугольников и один для объектов.

```
// Новые атрибуты для поддержки множественного отображения
```

```
// Многоугольник участвует во множественном отображении  
tfdefine POLY4DV2_ATTR_MIPMAP 0x0400
```

```
// Объект участвует во множественном отображении  
#define OBJECT4DV2_ATTR_MIPMAP 0x0008
```

Если загружается объект, участвующий во множественном отображении, мы должны установить соответствующий флаг в его атрибутах (мы делаем то же самое и в случае многоугольника, участвующего во множественном отображении). Но кто именно должен делать это? Итак, нужна еще одна функция загрузки объекта, поддерживающая множественное отображение. Я добавил эту функциональность в функцию чтения .COB-файлов Caligari, поскольку я использую ее чаще всего. Ниже представлен новый прототип функции.

```
int Load_OBJECT4DV2_COB2(  
    OBJECT4DV2_PTR obj,          // указатель на объект  
    char *filename,              // Имя .COB-файла  
    VECTOR4D_PTR scale,          // Начальное масштабирование  
    VECTOR4D_PTR pos,            // Начальное положение  
    VECTOR4D_PTR rot             // Начальный поворот  
    int vertex_flags,            // Флаги перегруппировки вершин  
                                // и выполнения преобразования  
    int mipmap ) '              // Флаг разрешения множественного  
                                // отображения  
                                // 0 означает отсутствие  
                                // множественного отображения,  
                                // 1 - его генерацию
```

Функция абсолютно идентична своей предыдущей версии, за исключением последнего параметра, представляющего собой флаг, указывающий, нужно ли выполнять множественное отображение. Если установить этот параметр равным 1 и у объекта есть текстуры, то они будут участвовать во множественном отображении. Если же этот флаг равен 0, то множественное отображение не будет выполняться. Функция загрузки просто гигантская, так что я не смогу привести ее листинг при всем моем желании. Однако в ней всего два новых блока кода. Первый из них устанавливает для атрибут множественного отображения для объекта.

```
// Перед выполнением текстурирования мы должны определить,  
// участвует ли текстура на этом объекте во множественном  
// отображении. Если да, то мы должны создать цепочку  
// множественного отображения и установить соответствующие  
// атрибуты объекта и многоугольников  
if (mipmap==1)  
{  
    // Устанавливаем бит множественного отображения объекта  
    SET_BIT(obj->attr, OBJECT4DV2_ATTR_MIPMAP);  
  
    // Теперь, имея базовую текстуру уровня d=0, вызываем  
    // генератор цепочки множественного отображения  
    Generate_Mipmaps(obj->texture,  
        (BITMAP_IMAGE_PTR *)&obj->texture);  
} // if
```

Затем, при окончательной установке атрибутов всех **многоугольников**, включается код для установки атрибутов множества **нного** отображения.

```
if (materials[poly_material[curr_poly]].attr &
    MATV1_ATTR_SHADE_MODE_TEXTURE)
{
    // Устанавливаем режим затенения
    SET_BIT(obj->plist[curr_poly].attr,
        POLY4DV2_ATTR_SHADE_MODE_TEXTURE);

    // Устанавливаем текстуру многоугольника
    obj->plist[curr_poly].texture = obj->texture;

    // Если объект участвует во множественном отображении,
    // указатель уже указывает на массив множественного
    // отображения; однако нужно установить соответствующим
    // образом атрибуты многоугольника
    if (mipmap)
        SET_BIT(obj->plist[curr_poly].attr,
            POLY4DV2_ATTR_MIPMAP);

    // Устанавливаем атрибуты координат текстуры
    SET_BIT(obj->vlist_local[obj->plist[curr_poly]
        .vert[0]].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
    SET_BIT(obj->vlist_local[obj->plist[curr_poly]
        .vert[1]].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
    SET_BIT(obj->vlist_local[obj->plist[curr_poly]
        .vert[2]].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
} // if
```

Вот и все изменения, которые вносятся в новый загрузчик объекта для создания и установки атрибутов множественного отображения. Теперь я хочу показать последнюю функцию, касающуюся множественного отображения, а именно функцию его удаления.

```
int Delete_Mipmaps(BITMAP_IMAGE_PTR *mipmaps,
    int leave_level_0)
{
    // Эта функция удаляет все текстуры цепочки
    // множественного отображения, на которые указывают
    // указатели в массиве множественного отображения. Затем
    // она освобождает выделенную память. Если флаг
    // leave_level_0 равен 1, функция оставляет нетронутым
    // верхний уровень текстур 0

    BITMAP_IMAGE_PTR*tmipmaps = (BITMAP_IMAGE_PTR*)*mipmaps;

    // Проверяем корректность указателя
    if (!tmipmaps) return(0);

    // Шаг 1: вычисляем общее число уровней множественного
    // отображения
```

```

int num_mip_levels = logbase2ofx[tmipmaps[0]->width]+1;

// Итеративно выполняем операцию удаления для каждой
// из текстур
for (int mip_level = 1; mip_level < num_mip_levels;
    mip_level++)
{
    // Освобождаем память буфера растрового изображения
    Destroy_Bitmap(tmipmaps[mip_level]);

    // Удаляем сам растровый объект
    free(tmipmaps[mip_level]);
} // mip_level

// Теперь, в зависимости от флага leave_level_0, либо
// удаляем все уровни, либо оставляем нетронутым
// уровень 0
if (leave_level_0 == 1)
{
    // Нам нужен временный указатель на растровое
    // изображение уровня 0
    BITMAP_IMAGE_PTR temp = tmipmaps[0];

    // Освобождаем *tmipmaps и присваиваем указатель
    // уровню 0
    free(*tmipmaps);
    *tmipmaps = temp;
} // if
else
{
    // Удаляем все!
    Destroy_Bitmap(tmipmaps[0]);

    // Сейчас удаляем сам растровый объект
    free(tmipmaps[0]);
} // else

// Возвращаем код успешного завершения
return (1);
} // Delete_Mipmaps

```

Функция `Delete_Mipmaps()` получает два параметра: указатель на указатель на растровое изображение и флаг необходимости удаления растрового изображения уровня 0. Причина, по которой нам нужен адрес указателя, аналогична причине передаче адреса указателя в функцию генерации цепочки множественного отображения — нам нужно изменить указатель при выходе из функции. Иными словами, нам нужно вернуть в прежнее состояние указатель на массив множественного отображения, который изначально указывал на нормальное растровое изображение `BITMAP_IMAGE`. Например, предположим, у вас есть объект `obj`, у которого есть одиночная текстура, отображаемая на объекте. На ее основании вы хотите построить цепочку множественного отображения с помощью вызова новой функции загрузки объекта, установив флаг разрешения множественного отображения равным 1. Функция получает текстуру и создает цепочку множественного отображения. После того

как работа сделана, нужно вернуть все в исходное состояние. Поэтому мы должны удалить **текстурные** элементы цепочки **множественного** отображения в диапазоне уровней от 1 до  $p$ , вернув указателю значение адреса текстурного элемента уровня 0.

Именно поэтому нам нужен адрес указателя, а не сам указатель. Функция `Delete_Mipmaps()` получает также параметр `leave_level_0`. Если он равен 1, текстура уровня 0 будет сохранена, а указатель получит соответствующее значение. Если `leave_level_0` равен 0, будет уничтожено все: массив множественного отображения, все текстуры, включая уровень 0, и указатели — все будет освобождено и приравнено к NULL.

## Выбор метода множественного отображения

Есть два способа множественного отображения при визуализации: с выводом отдельных пикселей или отдельных многоугольников. Пиксельный вывод использует специальную метрику или алгоритм для определения того, какое множественное отображение следует выбрать на основе информации о каждом пикселе. Для этого существует несколько методов, но все они слишком сложны, чтобы работать в реальном времени. Поэтому мы остановимся на методе множественного отображения с выводом отдельных многоугольников. Чтобы понять алгоритм выбора уровня множественного отображения, нужно вспомнить, почему мы вообще создаем уровни множественного отображения: чтобы минимизировать высокочастотные искажения изображения путем использования текстур меньших размеров. Нет смысла в использовании текстуры размером  $256 \times 256$  для многоугольника размером  $10 \times 10$  — при этом вы увидите все разнообразие артефактов, сверкание и т.п. Если же вы используете текстуру  $8 \times 8$ , это будет вполне корректный выбор. Таким образом, вы уже представляете, как должен работать алгоритм. Берется область проекции многоугольника (в экранных координатах) и сравнивается с областью множественного отображения уровня 0. Мы получим соотношение количества текстел и пикселей, которое должно быть максимально близко к 1:1. Иными словами, не следует использовать текстуру  $256 \times 256$  для многоугольника  $10 \times 10$ . Для него подойдет текстура  $8 \times 8$ . Аналогично, для многоугольника  $30 \times 30$  следует использовать текстуру  $32 \times 32$ , а не  $8 \times 8$ .

## Выбор уровня множественного отображения на основе соотношения количества текстел и пикселей

Теперь, когда у нас есть общий эвристический метод, посмотрим, нельзя ли создать надежную формулу, основанную на том факте, что на каждом уровне множественного отображения содержится четверть информации предыдущего уровня. **Каждый** раз, когда отношение количества текстел и пикселей превышает 4, нужно перемещаться по уровням цепочки множественного отображения вверх или вниз. Математически соотношение текстел и пикселей выражено в уравнении 12.5.

### Уравнение 12.5. Вычисление соотношения текстел и пикселей

---

$$\text{mip\_ratio} = \frac{\text{Площадь текстуры уровня } 0}{\text{Площадь проекции многоугольника}}$$

---

Например, наша текстура уровня 0 имеет размер  $256 \times 256$  (базовая текстура) и мы **текстурируем** многоугольник размером  $24 \times 90$ . При этом соотношение текстел и пикселей  $\text{mip\_ratio} = (256 \cdot 256) / (24 \cdot 90) \approx 30.3$ .

Мы знаем, что если это соотношение превышает 4, нужно переходить на следующий уровень цепочки множественного отображения. Чтобы найти требуемый уровень, нужно

вычислить  $\log_4 30.3$ , т.е. найти число  $x$ , для которого  $4^x = 30.3$ . К счастью, существует простая формула преобразования логарифмов от одного основания к другому, показанная в уравнении 12.6.

#### Уравнение 12.6. Изменение основания логарифма

$$\log_a x = \ln x / \ln a.$$

Здесь  $\ln x$  — это натуральный логарифм, или логарифм по основанию  $e = 2.718281828\dots$ .

Применяя эту формулу для вычисления  $\log_4 30.3$ , получим

$$\log_4 30.3 = \ln 30.3 / \ln 4 \approx 2.46.$$

Мы нашли требуемый уровень. Он равен 2.46, т.е. мы должны выбрать (после округления) второй уровень. Обратившись к табл. 12.4, можно найти, что второй уровень множественного отображения имеет размер  $64 \times 64$ , или общую площадь текстелей, равную 4096. Она не более чем в 4 раза отличается от реальной пиксельной площади проекции, равной 2160, поэтому мы действительно нашли наилучший размер текстуры, равный  $64 \times 64$  для многоугольника  $24 \times 90$ .

Чтобы убедиться, что все это работает, давайте рассмотрим еще пару небольших примеров. Если размер многоугольника равен  $200 \times 170$ , можно ожидать, что будет выбран уровень 0, поскольку текстура на этом уровне достаточно близка по размерам к многоугольнику. Проверим:

$$\text{mip\_ratio} = (256 \cdot 256) / (200 \cdot 170) \approx 1.927,$$

$$\log_4 1.927 \approx 0.473.$$

Округляя, получаем 0, т.е. текстуру размером  $256 \times 256$ . Давайте теперь попробуем еще один пример — на другом конце спектра. Предположим, у нас есть многоугольник размером  $3 \times 4$ . Можно ожидать, что будет выбран уровень 6 или 7, текстуры которых имеют размер  $4 \times 4$  или  $2 \times 2$ , соответственно.

$$\text{mip\_ratio} = (256 \cdot 256) / (3 \cdot 4) \approx 5461.33,$$

$$\log_4 5461.33 \approx 6.21.$$

Нам предлагается выбрать уровень 6. Как видите, метод действительно работает...

Единственная проблема, связанная с алгоритмом вычисления соотношения текстелей и пикселей, заключается в том, что мы должны вычислять площадь каждого многоугольника (треугольника), сравнивать ее с площадью текстуры, затем логарифмировать и искать уровень отображения... Хотя все это можно сделать с помощью нескольких команд процессора, в этом нет необходимости. Существует более легкий практический способ, который вполне работоспособен в 99% случаев и позволяет выполнять точную настройку. Он основан на линейном разбиении, которое мы сейчас и рассмотрим.

#### Упрощенный выбор уровня множественного отображения

Итак, нам надо уметь выбрать нужный из  $p$  различных уровней множественного отображения по мере уменьшения многоугольников. Обычно при удалении от наблюдателя они уменьшаются, не так ли? И крошечный многоугольник при приближении, и огромный многоугольник при удалении выглядят одинаково, но это — крайности. Главное, что мы можем использовать такой эвристический параметр, как расстояние, для выбора необходимого уровня.

На рис. 12.46 мы определяем максимальное расстояние, на котором выбирается последний уровень множественного отображения. Предположим, мы установили максимальное расстояние выбора уровня равным 10000 единиц. После этого для любого многоугольника, у которого координата  $z$  (максимальная или средняя) превышает 10000

единиц, будет всегда выбираться наивысший уровень (с размером текстуры 1x1). Для многоугольников в диапазоне 0-10000 единиц будет выбираться уровень, являющийся линейной функцией расстояния от точки обзора.



Рис. 12.46. Выбор уровня множественного отображения в зависимости от расстояния

Учитывая все сказанное, я хочу показать реальный код из функции контекста визуализации `Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16()`, которая выполняет выбор уровня множественного отображения.

```
// Проверяем, используется ли множественное отображение
if (rc->rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_MIPMAP)
{
    // Определяем, глобально ли множественное отображение
    if (rc->attr & RENDER_ATTR_MIPMAP)
    {
        // Определяем уровень множественного отображения для
        // данного многоугольника. Вначале определяем,
        // сколько уровней множественного отображения
        // имеется в цепочке для данного многоугольника
        int tmiplevels =
            logbase2ofx[((BITMAP_IMAGE_PTR *)
                (rc->rend_list->poly_ptrs[poly]
                    ->texture))[0]->width];

        // Теперь делим все расстояние на сегменты,
        // определяем, в каком сегменте находится
        // многоугольник и выбираем уровень

        // Позднее можно будет реализовать более сложный
        // алгоритм. Обратите внимание: я использовал
        // одиночную вершину. Вы можете захотеть найти
        // среднее значение - например, для длинных стенок,
        // расположенных перпендикулярно к наблюдателю,
        // наиболее значимой будет их средняя часть

        int miplevel - (tmiplevels * rc->rend_list
            ->poly_ptrs[poly]->tvlist[0].z / ->mip_dist);

        // Ограничиваем уровень отображения
        if (miplevel > tmiplevels) miplevel - tmiplevels;

        // На основе уровня выбираем текстуру
```

```

face.texture = ((BITMAP_IMAGE_PTR *)
    (rc->rend_list->poly_ptrs[poly]
    ->texture))[miplevel];
// Теперь нужно разделить каждую координату текстуры
// на 2 для каждого уровня отображения
for (int ts = 0; ts < miplevel; ts++)
{
    face.tvlist[0].u0 *= 0.5;
    face.tvlist[0].v0 *= 0.5;

    face.tvlist[1].u0 *= 0.5;
    face.tvlist[1].v0 *= 0.5;

    face.tvlist[2].u0 *= 0.5;
    face.tvlist[2].v0 *= 0.5;
} // for
} // if
else // Множественное отображение не глобально
{
    // В этом случае многоугольник участвует во
    // множественном отображении, однако вызывающая
    // функция не требует множественного отображения и
    // потому мы будем поддерживать его путем выбора
    // уровня 0, поскольку все равно указатель текстуры
    // указывает на цепочку множественного отображения
    face.texture = ((BITMAP_IMAGE_PTR *)
        (rc->rend_list->poly_ptrs[poly]->texture))[0];

    // Заметим, что манипуляции с текстурными
    // координатами не нужны
} // else
} // if
else
{
    // Присваиваем текстуру без изменений
    face.texture = rc->rend_list->poly_ptrs[poly]->texture;
} // if

```

Здесь много нового материала, поэтому начнем с самого начала. Фрагмент кода взят из одной из точек входа в **растеризатор**, в которой определяется, будет ли использоваться множественное отображение. При этом **рассматривается** множество разных случаев. Первый из них — когда многоугольник не имеет множественного отображения. Во втором случае многоугольник имеет множественное отображение, однако мы не работаем с ним. В последнем случае многоугольник участвует во множественном отображении, и мы выполняем его.

Давайте теперь пройдем по коду. Вначале вычисляется количество уровней логарифмированием ширины текстуры уровня 0.

```

int tmiplevels =
    logbase2ofx[((BITMAP_IMAGE_PTR *)
        (rc->rend_list->poly_ptrs[poly]->texture))[0]->width];

```

В этом фрагменте мы **ищем** ширину текстуры уровня 0; для этого нам, в частности, приходится преобразовывать указатель на массив указателей в указатель на текстуру. Если эта текстура имеет размер, например, 128x128, то мы получаем, что количество уров-

ней равно  $\log_2 128 = 7$ . После получения количества уровней мы выбираем текстуру с помощью следующего кода.

```
int miplevel = (tmiplevels*
rc->rend_list->poly_ptrs[poly]->tvlist[0].z/rc->mip_dist);
```

Выбор уровня осуществляется с использованием параметра `mip_dist` из контекста визуализации. Это переменная, которая определяет, какое расстояние делится на сегменты — иными словами, это расстояние, на котором выбирается текстура  $1 \times 1$ . Затем мы находим текстуру поверхности при помощи следующего кода.

```
face.texture = ((BITMAP_IMAGE_PTR *)
(rc->rend_list->poly_ptrs[poly]->texture))[miplevel];
```

Здесь опять приходится прибегнуть к массе отвратительных приведений указателя, поскольку указатель на текстуру на самом деле является указателем на массив указателей на текстуры.

Теперь — одна *очень* важная деталь, о которой я ранее не упоминал. Предположим, у нас есть координаты текстуры  $(u, v) = (63, 63)$  и уровень 0 размером  $64 \times 64$  пикселя. Тогда  $(63, 63)$  — это дальний угол текстуры. Однако при выборе другого уровня множественного отображения текстура уже не будет иметь размеры  $64 \times 64$ . Следовательно, для правильного отображения текстуры нам нужно уменьшить ее координаты (рис. 12.47).

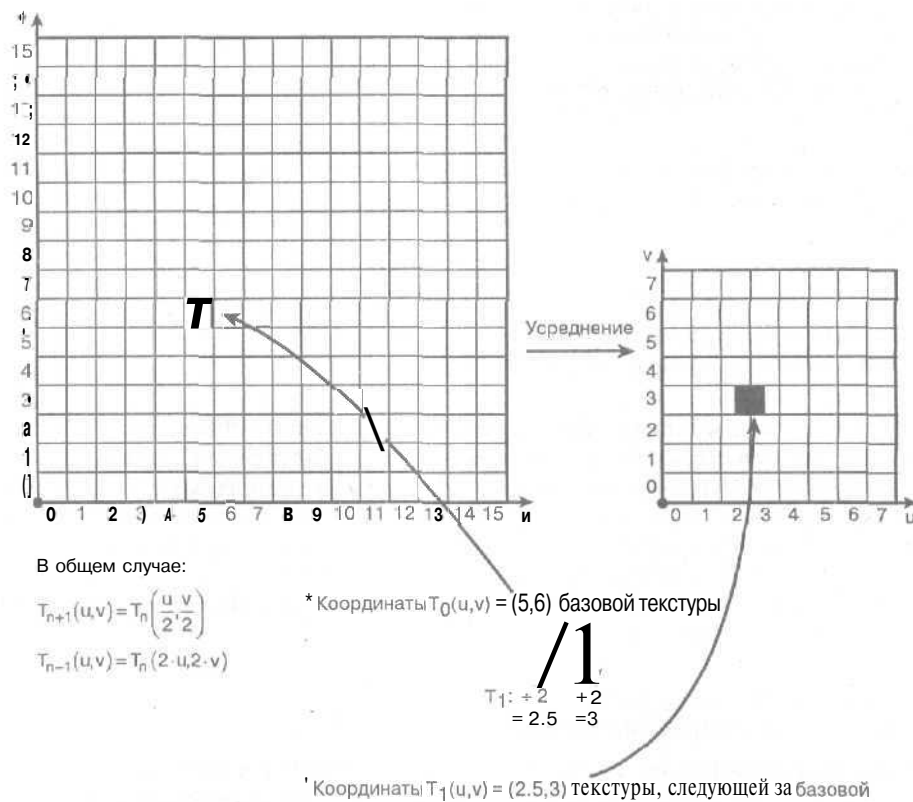


Рис. 12.47. Изменение координат текстуры в соответствии с уровнем отображения

Приведенный далее код определяет используемый уровень отображения и итеративно делит координаты текстуры на 2 (умножая их на 0.5) **соответствующее** количество раз,

```
for (int ts = 0; ts < miplevel; ts++)
{
    face.tvlist[0].u0* = 0.5;
    face.tvlist[0].v0* = 0.5;

    face.tvlist[1].u0* = 0.5;
    face.tvlist[1].v0* = 0.5;

    face.tvlist[2].u0* = 0.5;
    face.tvlist[2].v0* = 0.5;
} // for
```

Имеется гораздо более быстрый способ проделать все это, например, **использование** таблицы поиска с константами  $0.5^{\text{mip\_level}}$ , но сейчас нам важнее понимание, чем скорость.

#### НА ЗАМЕТКУ

Вся работа с координатами текстур, самими текстурами и т.д. выполняется с локальными переменными, так что беспокоиться о восстановлении каких-либо параметров нет необходимости.

Конечно, вы можете предпочесть метод вычисления отношения площадей рассмотренному более быстрому методу, это ваше **право**. А пока, перед тем как перейти к демонстрационной программе, давайте посмотрим на примеры кода загрузки объекта и настройки контекста визуализации. Типичный вызов для загрузки объекта с разрешением множественного отображения представлен ниже.

```
// Загружаем объект
Load_OBJECT4DV2_COB2(&obj_scene,
    "cube_flat_textured_01.cob",
    &vscale, &vpos, &vrot
    VERTEX_FLAGS_SWAP_YZ |
    VERTEX_FLAGS_TRANSFORM_LOCAL,
    1); // Включение множественного отображения
```

А вот пример настройки контекста и визуализации сцены.

```
rc.attr = RENDER_ATTR_ZBUFFER |
    RENDER_ATTR_MIPMAP |
    RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;
rc.video_buffer = back_buffer;
rc.lpitch = back_lpitch;
rc.mip_dist = -1500;
rc.zbuffer = (UCHAR *)zbuffer.zbuffer;
rc.zpitch = WINDOW_WIDTH*4;
rc.rend_list = &rend_list;
rc.texture_dist = 0;
rc.alpha_override = -1;

// Визуализация сцены
Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(&rc);
```

Демонстрационная программа DEMOII12\_11.CPP|EXE, копия экрана которой приведена на рис. 12.48, использует в своей работе множественное отображение и билинейную интерполяцию. Программа позволяет перемещаться в трехмерном пространстве и разрешать/запрещать множественное отображение.

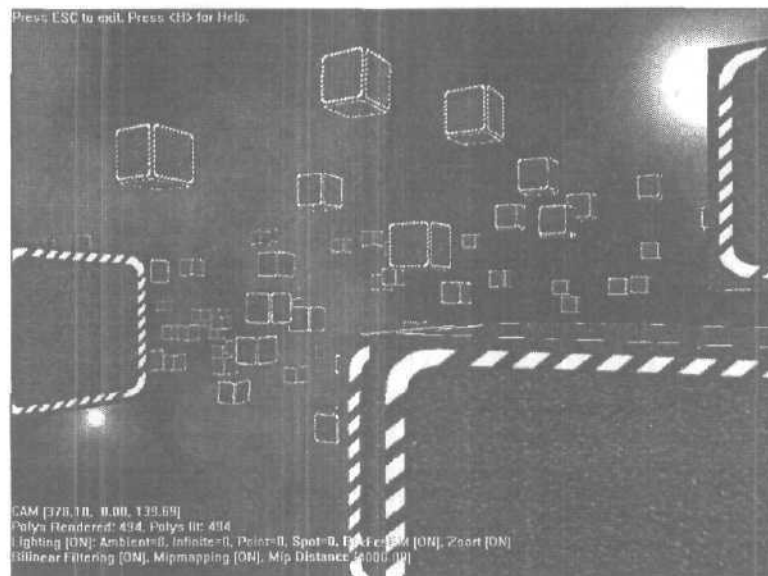


Рис. 12.48. Демонстрационная программа множественного отображения в действии

Обратите внимание, насколько лучше выглядят текстуры при множественном отображении. Кроме того, программа может выполнять билинейное **текстурирование**, поэтому картинка выглядит действительно хорошо. Клавиши управления программы:

- клавиши управления курсором — **перемещение**;
- <B> — включение/выключение билинейной **фильтрации**;
- <M> — включение/выключение множественного отображения;
- <1>, <2> — **уменьшение/увеличение** дистанции множественного отображения;
- <N> — выбор следующего объекта для экспериментов.

Не забудьте повозиться с расстоянием множественного отображения и посмотреть на изменение картинки на экране.

#### СОВЕТ

Используйте клавиши <1> и <2> для **изменения** дистанции множественного отображения. Сделайте ее равной примерно 500-1000, и вы увидите действительно эффектное множественное отображение.

Для компиляции программы требуются файлы DEMOII12\_11.CPP|H, модули T3DLIB1-10.CPP|H и библиотечные файлы DirectX.

## Трилинейная фильтрация

Трилинейная фильтрация использует билинейную интерполяцию вместе со множественным отображением. Вначале мы выбираем уровень множественного отображения для визуализации многоугольника или пикселя с помощью некоторого метода (вероятно, это будет метод площадей, поскольку он более точный) и получаем некоторое число наподобие 4.3. Оно говорит нам, что мы должны использовать уровни 4 и 5, т.е. выполнить интерполяцию от уровня 4 к **уровню 5** с помощью формулы

$\text{mip\_pixel} = (1-0.3) \cdot \text{mip\_level\_pixel\_4} + (0.3) \cdot \text{mip\_level\_pixel\_5}$

Таким образом, вместо использования одного уровня **множественного** отображения мы линейно интерполируем два. **Трилинейность** заключается в использовании третьей интерполяции — между уровнями — после нахождения пикселей на этих уровнях при помощи билинейной интерполяции (рис. 12.49).

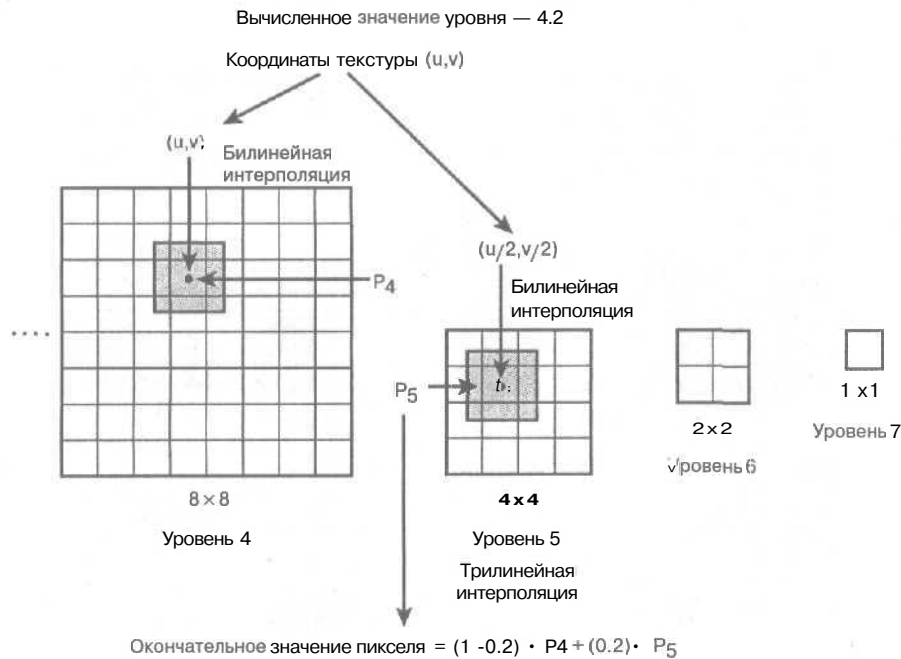


Рис. 12.49. Трилинейная фильтрация

На самом деле большинство программных или аппаратных процессоров не используют этот метод, который требует выполнения большого количества расчетов, не настолько улучшая изображение, чтобы мириться с замедлением работы. Однако вы можете добавить этот метод в наш процессор самостоятельно, просто для развлечения.

## Многопроходная визуализация и текстурирование

В это немного трудно поверить, но многопроходная визуализация с помощью программного растеризатора вполне возможна. Сейчас мы можем визуализировать примерно 1000–3000 многоугольников за кадр при скорости 15–30 fps, в зависимости от настроек процессора. Это значение можно уменьшить в 2 раза и выполнить при этом два прохода. **Многопроходность** позволяет осуществить массу всевозможных эффектов, например, эффект отражения в воде.

При многопроходной визуализации можно получать тени и другие специальные эффекты. Однако в нашем случае два прохода — это максимум, что можно попробовать **осуществить**, поскольку большее число проходов просто катастрофически скажется на производительности. В качестве примера многопроходной визуализации я создал сцену с **некоторыми**

простыми геометрическими формами, расположенными на плоскости. Для создания эффекта я выполняю иизуализацию плоскости без использования Z-буфера, а затем эффект отражения достигается визуализацией объектов с альфа-каналом и **Z-буфером** и инвертированной **матрицей** обзора. После этого я выполняю второй проход и визуализирую объекты **еще** раз, без альфа-канала и Z-буфера и с нормальной матрицей обзора. Окончательный результат — **блестящая** поверхность, отражающая объекты. На рис. 12.50 показана копия экрана этой демонстрационной программы.

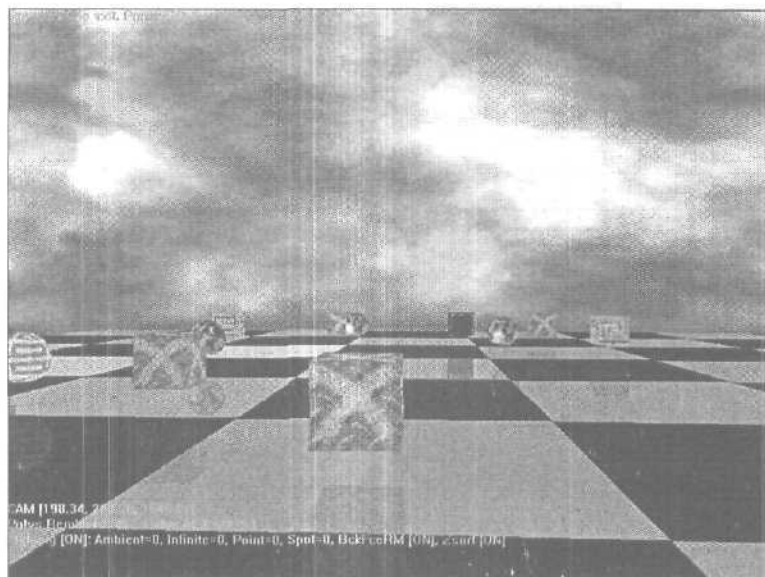


Рис. 12.50, Копия экрана демонстрационной программы, реализующей эффект отражения

Программа находится в файлах DEMO112\_12.CPP|EXE. Клавиши управления курсором позволяют перемещаться в пространстве, клавиша <P> —переключать режим многопроходное™, что позволяет увидеть процесс построения сцены. Для компиляции программы необходимы файлы DEMO112\_12.CPP|H, модули T3DLIB1-10.CPP|H, а также библиотечные файлы DirectX.

## Все в одном вызове

Как видим, размер кода становится угрожающим и выходит из под контроля, а нам нужно добавить еще один уровень косвенности поверх имеющихся вызовов функций для вывода списка визуализации. Сейчас у нас есть различные версии **растеризаторов** — с сортировкой, Z-буфером, 1/z-буфером, множественным отображением, альфа-каналом, перспективой и т.д. Их очень много, что попросту неприемлемо для нашего игрового процессора. Взгляните на рис. 12.51, который должен пояснить вам, о чем я говорю. Конечно, если вы точно знаете, какой вид **растеризатора** вам нужен, вам не требуется еще один уровень косвенности, но это бывает не всегда, и создание даже простой демонстрационной программы при наличии такого разнообразия функций представляет собой довольно сложную задачу.

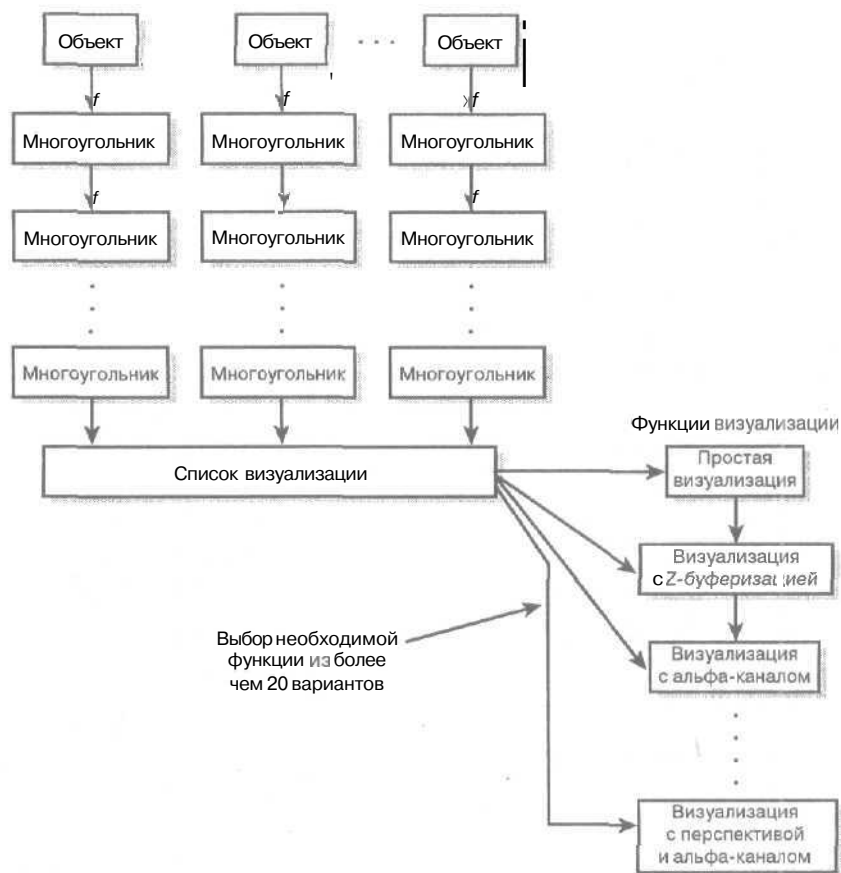


Рис. 12.51. Сложность выбора функции визуализации

## Новый контекст визуализации

Для упорядочения вызовов я создал новую структуру данных, именуемую *контекстом визуализации*, которую можно настраивать наподобие структуры DirectX. Цель этой структуры данных в заполнении ее информацией, такой как видеобуфер, Z-буфер, флаги визуализации и т.п., чтобы иметь возможность *осуществить* единый вызов *высокоуровневой* функции, которая анализирует контекст визуализации и выполняет все необходимые вызовы для конкретного треугольника. На рис. 12.52 показана абстрактная схема данного решения.

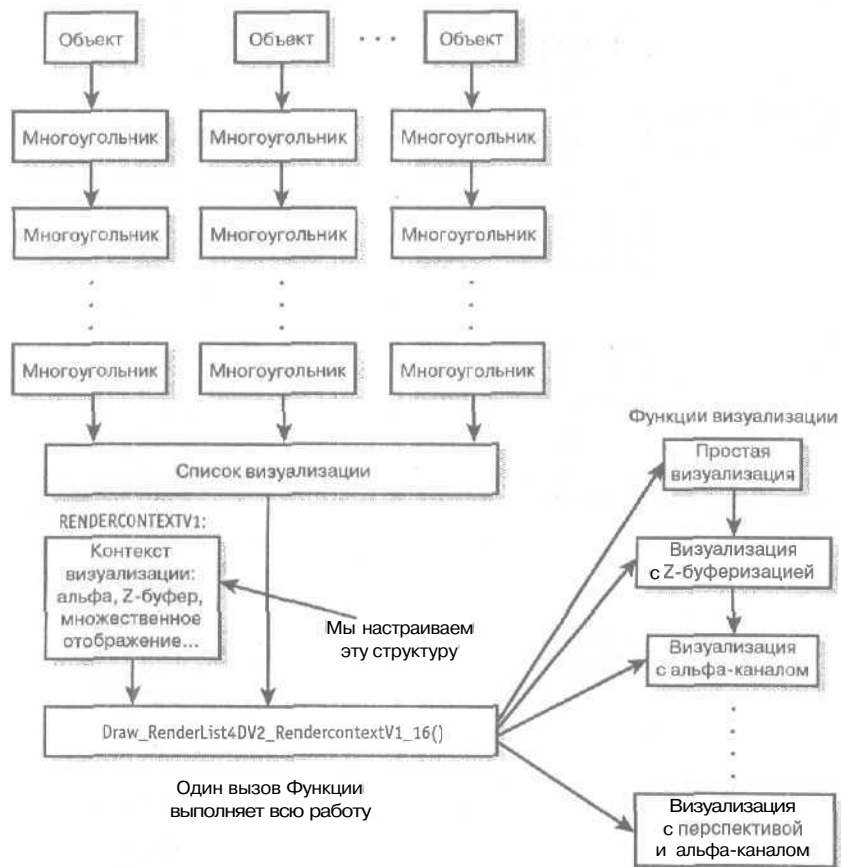


Рис. 12.52. Система с поддержкой контекста визуализации

Ниже представлена новая структура данных (которую можно найти в файле T3DLIB10.H).

// Этот новый тип содержит контекст визуализации, благодаря  
 // которому нам не нужно передавать все новые и новые  
 // параметры в функции визуализации. Мы можем заполнить эту  
 // структуру нужной информацией и затем передать ее целиком

```

typedef struct RENDERCONTEXTV1_TYP
{
    int attr; // Атрибуты визуализации
    RENDERLIST4DV2_PTR rend_list; // Указатель на список визуализации
    UCHAR *video_buffer; // Указатель на видеобуфер
    int lpitch; // Шаг памяти видеобуфера в байтах
    UCHAR *zbuffer; // Указатель на Z-буфер или 1/z-буфер
    int zpitch; // Шаг памяти г- или 1/z-буфера в байтах
    int alpha_override; // Глобальное значение альфа
    int mip_dist; // Максимальное расстояние для

```

```

// деления на уровни множественного
// отображения
int texture_dist // Расстояние, для которого
texture_dist2; // разрешено аффинное
// текстурирование при гибридном
// режиме перспективного и
// аффинного текстурирования

// Будущие расширения
int ival1, ival2; // Дополнительные целочисленные
// поля
float fval1, fval2; // Дополнительные поля с плавающей
// точкой
void *vptr; // Дополнительный указатель
} RENDERCONTEXTV1, *RENDERCONTEXTV1_PTR;

```

По сути, здесь представлен список параметров объединения всех функций визуализации. В нем есть все, что может понадобиться для настройки визуализации. Все поля структуры имеют **интуитивно-понятные** имена и означают то, что и обычно. Тем не менее, я хочу их перечислить: чтобы убедиться, что мы с вами находимся на одной и той же странице :-)

- `attr` — этот параметр управляет тем, как именно выполняется визуализация (с Z-буферизацией или без нее, с альфа-смешиванием, билинейной интерполяцией, множественным отображением и т.д.). Перед выполнением вызова функции визуализации необходимо присвоить данной переменной необходимое значение, объединив необходимые значения с использованием побитового **ИЛИ**;
- `rend_list` — указатель на список визуализации;
- `video_buffer` — указатель на видеобuffer, в котором выполняется визуализация сцены;
- `lpitch` — шаг памяти видеобufferа в байтах;
- `zbuffer` — указатель на хранилище Z-буфера. Вы можете использовать Z-буфер или инверсный Z-буфер, но хранилище должно быть всегда одно и то же;
- `zpitch` — шаг памяти хранилища Z-буфера в байтах;
- `alpha_override` — если вы разрешили альфа-смешивание в контексте визуализации, то можете изменить параметр альфа для всей сцены путем помещения в эту переменную целого числа из диапазона (0..NUM\_ALPHA\_LEVELS-1). Если каждый объект должен иметь свое значение параметра альфа, установите величину `alpha_override` равной `-1` ;
- `mip_dist` — этот параметр указывает величину диапазона, в котором выполняется множественное отображение. В пределах диапазона расстояний от точки обзора от 0 до `mip_dist` уровень множественного отображения линейно зависит от расстояния;
- `texture_dist`, `texture_dist2` — эти параметры используются для управления диапазоном в гибридных режимах **текстурирования**. Вы должны указать один или оба параметра в зависимости от выбранного гибридного режима;
- `ival1`, `ival2` — это просто дополнительные целочисленные переменные, которые вы можете использовать по своему усмотрению;
- `fval1`, `fval2` — это просто дополнительные переменные с плавающей точкой, которые вы можете использовать по своему усмотрению;
- `vptr` — **указатель типа `void*`, который вы можете использовать по своему усмотрению.**

## Настройка контекста визуализации

Вначале контекст визуализации нужно создать (обычно это одиночная глобальная переменная, но можно создать и несколько контекстов).

```
RENDERCONTEXTV1 rc;
```

Для очистки памяти любой большой структуры рекомендуется использовать следующий код.

```
memset(&rc,0,sizeof(rc));
```

Поле attr контекста визуализации является ключом ко всему — нужно просто указать в нем нужные опции визуализации. Соответствующие флаги находятся в `T3DLIB10.H` и перечислены ниже.

```
// Макроопределения атрибутов, управляющих работой функции
// визуализации. Заметим, что везде, где это возможно,
// каждый класс флагов управления располагается в 4-битном
// слове. Это помогает при последующем расширении
// возможностей функций

// Без Z-буферизации, многоугольники визуализируются в
// соответствии со списком
#define RENDER_ATTR_NOBUFFER          0x00000001

// Растеризация с использованием Z-буферизации
#define RENDER_ATTR_ZBUFFER           0x00000002

// Растеризация с использованием 1/z-буферизации
#define RENDER_ATTR_INVZBUFFER        0x00000004

// Использование множественного отображения
#define RENDER_ATTR_MIPMAP             0x00000010

// Разрешаем альфа-смешивание
#define RENDER_ATTR_ALPHA              0x00000020

// Разрешаем билинейную фильтрацию, но только для
// постоянного затенения и аффинных текстур
#define RENDER_ATTR_BILERP            0x00000040

// Используем аффинное текстурирование для всех
// многоугольников
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE 0x00000100

// Используем текстурирование с точной перспективой
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT 0x00000200

// Используем текстурирование с линейно-кусочной
// перспективой
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR 0x00000400
```

```
// Используем гибридное аффинное и линейно-кусочное
// текстурирование, основанное на расстоянии
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1 0x00000800
```

```
// Еще не реализовано
ftdefine RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID2 0x00001000
```

Теперь давайте рассмотрим некоторые примеры. Предположим, что вы хотите настроить контекст визуализации, в котором вообще нет буферизации (вы выполняете Z-сортировку многоугольников самостоятельно или каким-то иным способом).

### Настройка контекста визуализации: пример 1

Без альфа-смешивания, без множественного отображения, но с аффинным текстурированием.

```
rc.attr = (RENDER_ATTR_NOBUFFER |
           RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE);
rc.video_buffer = back_buffer; // Видеобуфер
rc.lpitch = back_lpitch;
rc.mip_dist = 0;
rc.zbuffer = NULL;
rc.zpitch = 0;
rc.rend_list = &rend_list; // Список визуализации
rc.texture_dist = 0;
rc.alpha_override = -1;
```

### Настройка контекста визуализации: пример 2

z-буферизация с поддержкой альфа-смешивания и линейно-кусочной перспективой.

```
rc.attr = (RENDER_ATTR_ZBUFFER |
           RENDER_ATTR_ALPHA |
           RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR);
rc.video_buffer = back_buffer; // Видеобуфер
rc.lpitch = back_lpitch;
rc.mip_dist = 0;
```

```
// Теперь этот параметр нужен
rc.zbuffer = (UCHAR *)zbuffer.zbuffer;
rc.zpitch = WINDOW_WIDTH*4;
rc.rend_list = &rend_list;
rc.texture_dist = 0;
rc.alpha_override = -1;
```

### Настройка контекста визуализации: пример 3

Теперь давайте попробуем настроить контекст для множественного отображения для диапазона в 2000 единиц. Конечно, требуются текстурирование с корректной перспективой и I/z-буфером.

```
rc.attr = (RENDER_ATTR_INVZBUFFER |
           RENDER_ATTR_ALPHA |
           RENDER_ATTR_MIPMAP |
           RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT);
rc.video_buffer = back_buffer; // Видеобуфер
```

```
rc.lpitch - back_lpitch;
rc.mip_dist = 2000; // Дистанция
rc.zbuffer =
    (UCHAR *)zbuffer.zbuffer; // l/z-буфер
rc.zpitch = WINDOW_WIDTH*4;
rc.rendlist = &rend_list;
rc.texture_dist = 0;
rc.alpha_override = -1;
```

#### НА ЗАМЕТКУ

Вспомните, что переменная `mip_dist` определяет расстояние от точки обзора, в пределах которого уровни цепочки множественного отображения линейно зависят от расстояния.

## Настройка контекста визуализации: пример 4

Еще один пример использует гибридное **текстурирование**. Выбираем режим 1, который является **линейно-кусочным** режимом на ближней дистанции и переключается в **аффинный** режим после расстояния `texture_dist`. Мы установим это расстояние равным 500 единицам. Все объекты в диапазоне 0–500 будут использовать **линейно-кусочное текстурирование**, а за его пределами — **аффинное текстурирование**. Остальные параметры остаются теми же, что и в предыдущем примере.

```
rc.attr = (RENDER_ATTR_INVZBUFFER |
    RENDER_ATTR_ALPHA |
    RENDER_ATTR_MIPMAP |
    RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1);
rc.video_buffer = back_buffer; // Videобуфер
rc.lpitch - back_lpitch;
rc.mip_dist = 2000;
rc.zbuffer =
    (UCHAR *)zbuffer.zbuffer; // l/z-буфер
rc.zpitch - WINDOW_WIDTH*4;
rc.rend_list = &rend_list;
rc.texture_dist = 500;
rc.alpha_override = -1;
```

#### ВНИМАНИЕ

При использовании любой формы **текстурирования** с корректной **перспективой** вы обязаны использовать **l/z-буфер**, поскольку такое **текстурирование** не поддерживается простым **растеризатором** или **растеризатором с Z-буфером**.

## Вызов функции визуализации

Теперь, после настройки контекста визуализации пришло время заняться самой визуализацией. Нам нужна функция, которая получает контекст визуализации и выполняет всю работу за нас, самостоятельно определяя, какой **растеризатор** следует вызывать для каждого из многоугольников. В этой функции множество условных операторов, но я все равно хочу, чтобы вы познакомились с ней.

Следует заметить, что существуют и лучшие способы решения поставленной задачи. Так, например, у меня во внутреннем цикле используются условные операторы, которые вполне можно было вычислить предварительно. Словом, здесь есть что оптимизировать, так что можете заняться этим на досуге.

Главная проблема, связанная с этой функцией, заключается в том, что она довольно большая (15–30 страниц), поэтому я не могу привести ее листинг. Но вы можете найти ее

на прилагаемом компакт-диске в файле `T3DLIB10.CPP`. Я очень прошу вас не пожалеть времени и внимательно познакомиться с ней.

Функция немного оптимизирована, и сначала определяет, будет ли выполняться визуализация без использования буфера, с использованием *Z-буфера* или *1/z-буфера*, так что при ознакомлении в принципе можно ограничиться только первой частью — визуализацией без буферизации.

Вызов этой функции выполняется следующим образом.

```
Draw_RENDERLISTV2_RENDERCONTEXTV1_16(&rc);
```

Правда, это стоит затраченных усилий?

## Резюме

Здесь мы написали больше кода, чем в любой другой главе, но все это стоит затраченных усилий. Теперь игровой процессор поддерживает столько возможностей, что он уже готов для создания игр. Мы добавили в него *текстурирование* с корректной перспективой, альфа-смешивание, множественное отображение, *1/z-буферизацию*, множество видов *оптимизации* и создали единую функцию растеризации. Кроме того, мы добавили новые возможности в загрузчик *.SOB-файлов* и в генератор ландшафтов. Не последнюю роль играют хорошие демонстрационные программы, которые могут послужить отправными точками для ваших экспериментов. Помните, основная цель этой книги — не научить самым современным методам, а дать прочное понимание трехмерных алгоритмов и их реализации в реальном времени. Забавно, что при всем этом игровой процессор получился довольно хорошим — когда одна из таких программ работала на моем компьютере, один сотрудник зашел ко мне и решил, что запущена программа с аппаратной поддержкой, поскольку в ней были эффекты освещенности и текстурирования. Даже если завершить книгу прямо сейчас, у вас уже есть более чем достаточно знаний для того, чтобы продолжать работать над созданием трехмерных игр.

Единственное, что вызывает сожаление, — в коде уже есть ошибки, иногда проявляющиеся на системном уровне. Увы, но уже написано столько кода, что мне просто трудно (да и некогда) его тестировать. В конце концов, цель книги и демонстрационных программ — это ваше обучение, поэтому вы можете написать свой собственный игровой процессор и сделать собственные ошибки.

Далее мы рассмотрим методы пространственного разделения и приступим к работе с геометрией игрового мира.



# ГЛАВА 13

## Алгоритмы разбиения пространства и определения видимости

### В этой главе...

• Новый модуль игрового процессора	1144
• Разбиение пространства и определение видимости	1144
• Двоичное разбиение пространства	1148
• Потенциально видимые множества	1228
• Порталы	1237
• Ограничивающие иерархические объемы и октадеревья	1242
• Отбор с учетом препятствий	1262

В данной главе мы рассмотрим алгоритмы разбиения пространства, такие как двоичное разбиение пространства, ограничивающие иерархические объемы, октадеревья, порталы и т.п. Кроме того, мы обсудим общие алгоритмы определения видимости, такие как потенциально видимые множества и отбраковка преград. Из-за **высокой** сложности и значительного объема материала мы реализуем лишь несколько методов, включая **BSP-деревья**. Однако и этого будет вполне достаточно. Ниже перечислены основные темы, рассматриваемые в данной главе:

- новый библиотечный модуль игрового процессора;
- разбиение пространства и определение видимых поверхностей;

- двоичное разбиение пространства;
- порталы;
- ограничивающие иерархические объемы и **октадеревья**;
- отбраковка преград.

## Новый модуль игрового процессора

Как и ранее, в этой главе будет написано много кода, который можно объединить в отдельный библиотечный модуль. Мы добрались до создания модуля **T3DLIB11**, поэтому для **компиляции** любой программы из этой главы потребуется основной **.CPP-файл**, библиотечные файлы **DirectX**, а также новые библиотечные модули:

**T3DLIB11.CPP** — исходный файл на языке C/C++ для методов разбиения пространства и т.п.;  
**T3DLIB11.H** — соответствующий заголовочный файл.

НА ЗАМЕТКУ

Самой собой разумеется, что для **компиляции** нужны остальные библиотечные модули **T3DLIB1-10.CPP;H**.

Я бы хотел полностью привести исходные тексты библиотеки и заголовочного файла, но я вынужден экономно использовать объем книги. Тем не менее, в тексте будут представлены все функции и структуры данных — если не **полностью**, то хотя бы их прототипы.

## Разбиение пространства и определение видимости

Разбиение пространства и определение видимых поверхностей тесно взаимосвязаны. В общем случае любой заданный трехмерный мир наполнен объектами, ландшафтами, интерьерами и открытыми пейзажами. Проблема в том, что все эти сетки модели могут состоять из тысяч, миллионов (а в ближайшем будущем — и миллиардов) многоугольников. До сих пор мы напряженно работали над поиском способов удаления максимально возможного количества многоугольников из конвейера визуализации. Полученная система главным образом основана на объектах, причем каждый объект представляет собой набор многоугольников. Эти объекты преобразуются и затем включаются в глобальный список визуализации для дальнейших преобразований, таких как освещение, отсечение, **проецирование** и **растеризация**. Как уже **отмечалось**, можно исключать из области видимости целые объекты, ограничивая каждый объект сферой и исключая его из области видимости до отправки объекта в конвейер визуализации, как показано на рис. 13.1.

Однако многие объекты не удастся отбросить с помощью такого грубого метода, и они попадают в конвейер визуализации. Например, длинные объекты, или объекты, состоящие из многих частей и т.п., плохо вписываются в ограничивающие сферы. Но мы на этом не останавливались, и выполняли этап удаления задних поверхностей объекта, в ходе которого из конвейера удаляются многоугольники, которые не видны **наблюдателю**. Эти методы показаны на рис. 13.2.

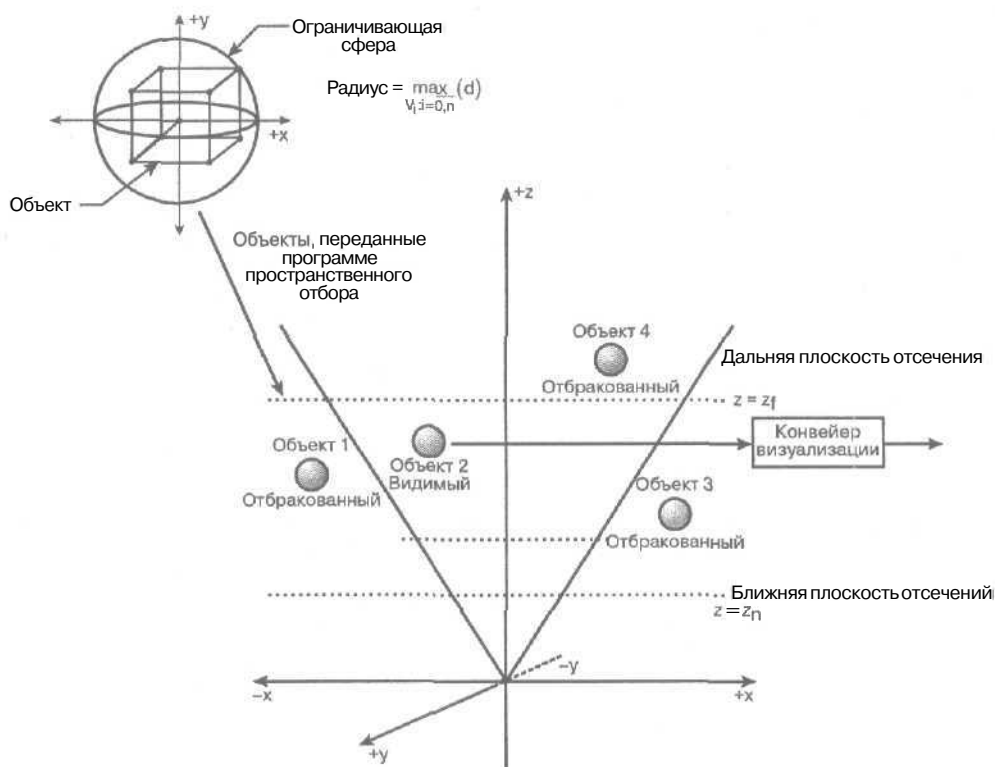


Рис. 13.1. Отбор с помощью ограничивающих сфер

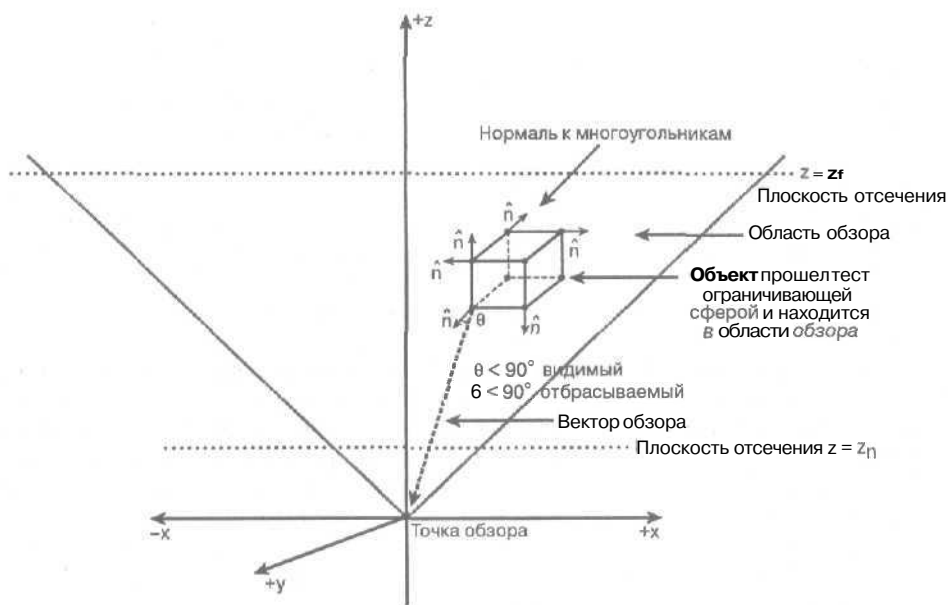


Рис. 13.2. Исключение задней поверхности

Наконец, многоугольники, которые не удалось отбросить, отсекаются областью обзора в пространстве объектов; окончательное отсечение производится в двумерном пространстве в процессе растеризации. Процесс отсечения является формой отбраковки на основе многоугольников, в ходе которого могут отсекаться целые многоугольники, как показано на рис. 13.3.

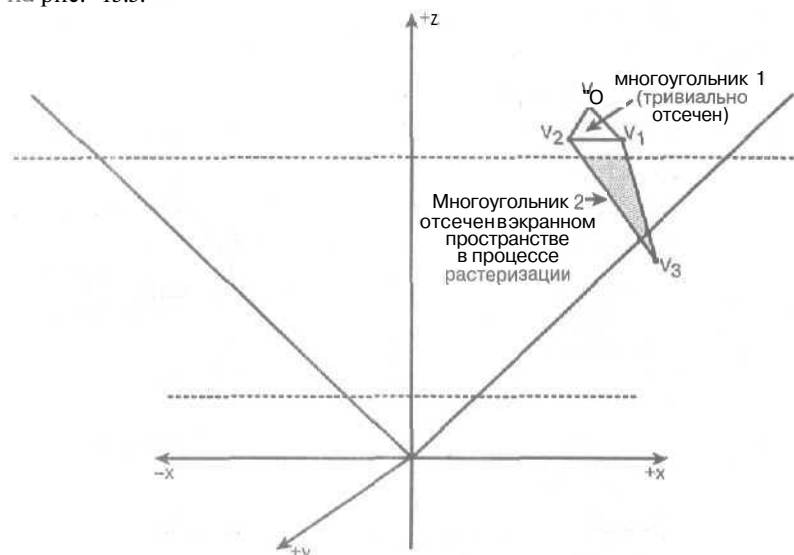


Рис. 13.3. Отсечение многоугольника

Данная система хорошо работает для открытых пространств со сравнительно небольшим количеством объектов и даже с нашими примитивными демонстрационными версиями местности. Проблема в том, что мы не используем геометрические закономерности, **вытекающие** из геометрии ячеек или здравого смысла. Это и есть тема данной главы. Мы хотим решить две **первоочередные** задачи.

Задача 1. Как осуществить разбиение пространства/объектов (**возможно**, используя предварительные вычисления или какие-то допущения или предположения об игровом пространстве или объектах), чтобы впоследствии можно было быстро отбрасывать значительные части сцены и тем самым облегчить визуализацию и определение видимости?

Задача 2. Существуют ли способы, позволяющие определить, какие многоугольники **являются** видимыми из данной точки наблюдения в игре, и обрабатывать только те многоугольники, которые **потенциально** видимы из указанной точки?

Эти задачи несколько перекрываются и являются составной частью рассматриваемой в данной главе темы определения видимости, преград и разбиения пространства.

Главной проблемой компьютерной **графики** является не растеризация, освещенность или другие рассмотренные ранее темы. Все это, конечно, важно и, кстати говоря, в основном указанные проблемы решены на аппаратном уровне. Задачи же, о которых пойдет речь в данной главе, аппаратно не решены и, вероятно, никогда и не будут решены. Иными словами, любой отдельно взятый метод будет работать с конечным числом многоугольников, но при увеличении этого числа в десятки раз понадобится новый метод. Алгоритм рисования линии универсален — не имеет значения, что мы рисуем, лошадь или космический корабль, однако алгоритмы разбиения пространства и определения видимости зависят от типа и размеров множеств данных, с которыми они работают. По мере того, как мы создаем все более сложные миры, эти множества данных становятся все

больше и больше. Поэтому проблема определения видимости и отбраковки объектов приобретает первостепенное значение, поскольку не визуализировать некий объект — это самый быстрый способ нарисовать его.

Предположим, что мы хотим использовать нашу функцию генерации ландшафтов для создания местности  $1024 \times 1024$ . Это можно сделать, но тогда, поскольку местность является единым объектом, отбору и отсечению будет подвергаться огромное количество многоугольников. Более удачным способом является создание иерархии объектов местности в мире, поделенном на ячейки размером  $32 \times 32$ , где каждая ячейка представляет собой ландшафт из  $32 \times 32$  элементов (или  $32 \times 32 \times 2$  многоугольников, поскольку на каждый элемент приходится по два треугольника). Это позволит нам **отбраковывать** каждую ячейку местности как единый объект (рис. 13.4).

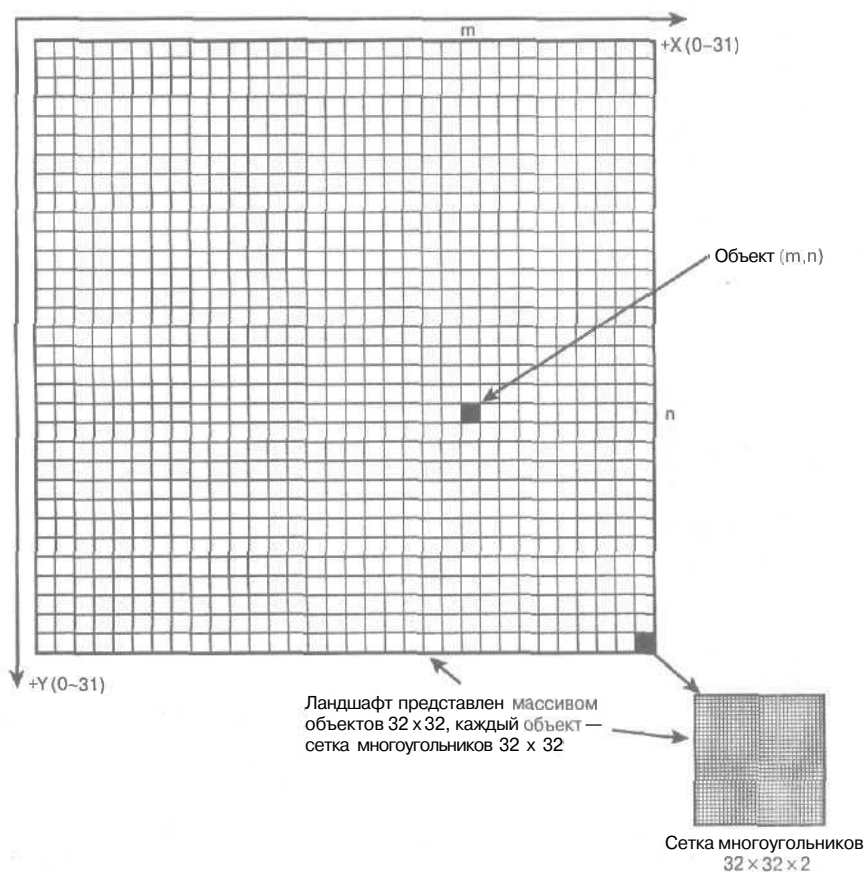


Рис. 13.4. Представление иерархии объектов

Можно ли предложить более удачный метод? Например, выбрать размер ячеек местности  $8 \times 8$  и поместить вокруг них ограничивающие прямоугольники, или использовать ячейки размерами  $16 \times 16$  или  $32 \times 32$ ? Таким образом можно создавать иерархические структуры данных о местности, содержащие больше или меньше данных, а затем, на этапе отбраковки или определения видимости, мгновенно отбрасывать огромные количества данных. Это абсолютно необходимо при описании больших территорий, закрытых помещений и космиче-

ских миров. Представьте себе задачу моделирования галактики, содержащей 100000000 звездных систем. А это действительно возможно: можно запомнить положение планет в каждой звездной системе, можно составить алгоритм с набором **параметров**, на лету генерирующий ландшафт планеты с любым заданным уровнем детализации, и создать список строений, растительности и т.п. для каждой планеты. Однако в действительности мы не в состоянии на **каждой** итерации отслеживать все 100000000 звездных систем и **определять**, какие из них должны визуализироваться. В то же время с помощью системы разбиения пространства это можно осуществить всего за несколько итераций.

Мы рассмотрим несколько методов разбиения пространства, начиная с метода двоичного разбиения, который является одним из наиболее часто используемых и мощных алгоритмов, поэтому в данной главе будет приведена его реализация. Вспомните, однако, почему мы нуждаемся в данных методах: при работе с огромными множествами данных мы не можем просто отправлять многоугольники в конвейер визуализации.

Кроме того, в процессе работы над обычной игрой-“стрелялкой” с одним игроком 99% времени мы находимся на уровне с сотнями комнат, но видеть мы можем только геометрию текущей комнаты, в которой расположена точка наблюдения (впрочем, конечно, в комнате могут быть окна). Естественно, нам не хотелось бы визуализировать все лишние комнаты, и потому необходимо найти методы и алгоритмы для решения проблем наподобие той, что проиллюстрирована на рис. 13.5.



Рис. 13.5. Проблема определения видимости

Мы рассмотрим несколько методов (некоторые только теоретически, другие — практически), после чего вы сможете самостоятельно решать задачи внутренней/внешней визуализации или, по крайней мере, будете **знать**, что необходимо делать и с чего начать.

## Двоичное разбиение пространства

Метод *двоичного разбиения пространства* (binary space partition, BSP) принадлежит к классу алгоритмов разбиения трехмерного пространства, в которых быстрота работы достигается за

счет разбиения исходного пространства и проведения предварительных вычислений (при наличии определенных геометрических ограничений). По сути на вход метода поступает набор многоугольников, а затем с помощью рекурсивного алгоритма создается двоичное дерево, обладающее рядом интересных свойств. Эти свойства позволяют совершать обход дерева в точном порядке от более удаленных к менее удаленным или от менее удаленных к более удаленным многоугольникам. Данное дерево можно также использовать при обнаружении столкновений или крупномасштабной отбраковки. На этой структуре данных построены такие игры, как *Doom*, *Quake* и другие. **BSP-дерево** разбивает пространство на выпуклые подпространства, используя **разделяющие** плоскости, параллельные осям или самим многоугольникам (главное, что для разбиения пространства используется его геометрия).

Рассмотрим, как работает метод **BSP**. На рис. 13.6 представлен вид сверху в плоскости  $xz$ , который будет использоваться в качестве иллюстрации в последующем объяснении. На рисунке изображено множество многоугольников, которое может задавать геометрию некоего уровня игры. Каждый многоугольник определяет плоскость, в которой он лежит. Если поместить наблюдателя в любой точке внутри рисунка, несложно вручную определить порядок рисования многоугольников при любом направлении наблюдения для правильной визуализации сцены,



Рис. 13.6. Задача, которую предстоит решить с помощью **BSP**

Например, на рис. 13.7 порядок визуализации должен быть от дальних к ближним: а, б, с, d, e, f. Это та же задача, которую мы уже решали ранее с помощью алгоритма художника и Z-буферизации. Однако эту задачу можно решить и иным способом, с использованием **BSP-дерева**. Для любой точки наблюдения метод **BSP** позволяет указать порядок следования многоугольников от дальних к ближним и наоборот, а также предоставляет множество другой полезной информации.

Алгоритм **BSP** работает следующим образом: берутся все многоугольники сетки или среды и производится их разбиение на выпуклые подпространства с помощью разделительных плоскостей. Эти разделительные плоскости могут быть параллельными осям,

произвольными или компланарными многоугольникам. Бегло ознакомимся с каждым из этих методов, а затем остановимся на наиболее употребительном.

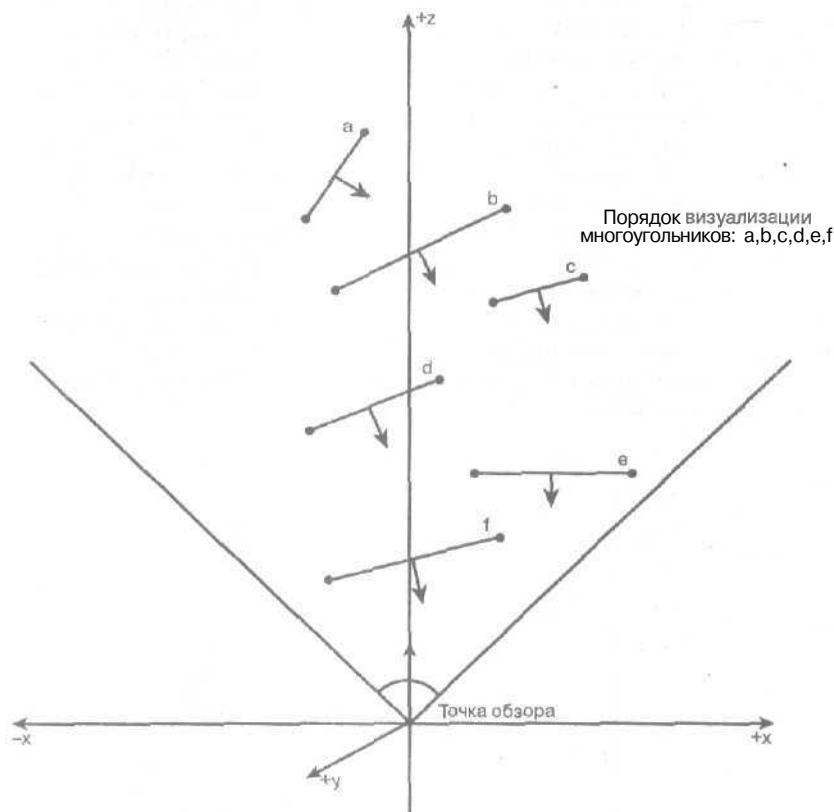


Рис. 13.7. Порядок визуализации объектов при визуализации от дальних к ближним

## Двоичное разбиение пространства плоскостями, параллельными осям

На рис. 13.8 представлен исходный набор многоугольников и последующее разбиение этих многоугольников плоскостями, параллельными осям координат. Как видите, пространство разбивается на небольшие прямоугольники так, чтобы **каждый** прямоугольник содержал единственный многоугольник. Если разделительная плоскость пересекает некий многоугольник, этот многоугольник расщепляется и в большинстве случаев добавляется к обоим подпространствам, образованным данной разделительной плоскостью. Однако можно использовать и другой метод: снабдить данный многоугольник в обоих подпространствах (или прямоугольниках, как в данном случае) специальной меткой, а позднее во время работы пометить его как обработанный. При попытке повторно получить доступ к данному многоугольнику при обходе другого подпространства он игнорируется, поскольку он уже учтен и обработан. Кроме того, заметим, что, по определению, каждое полученное в результате разбиения пространство является выпуклым, поскольку любой квадрат или куб тоже является выпуклым. Таким образом, каждый узел BSP определяет выпуклое подпространство; к этому свойству мы **еще** вернемся позднее.

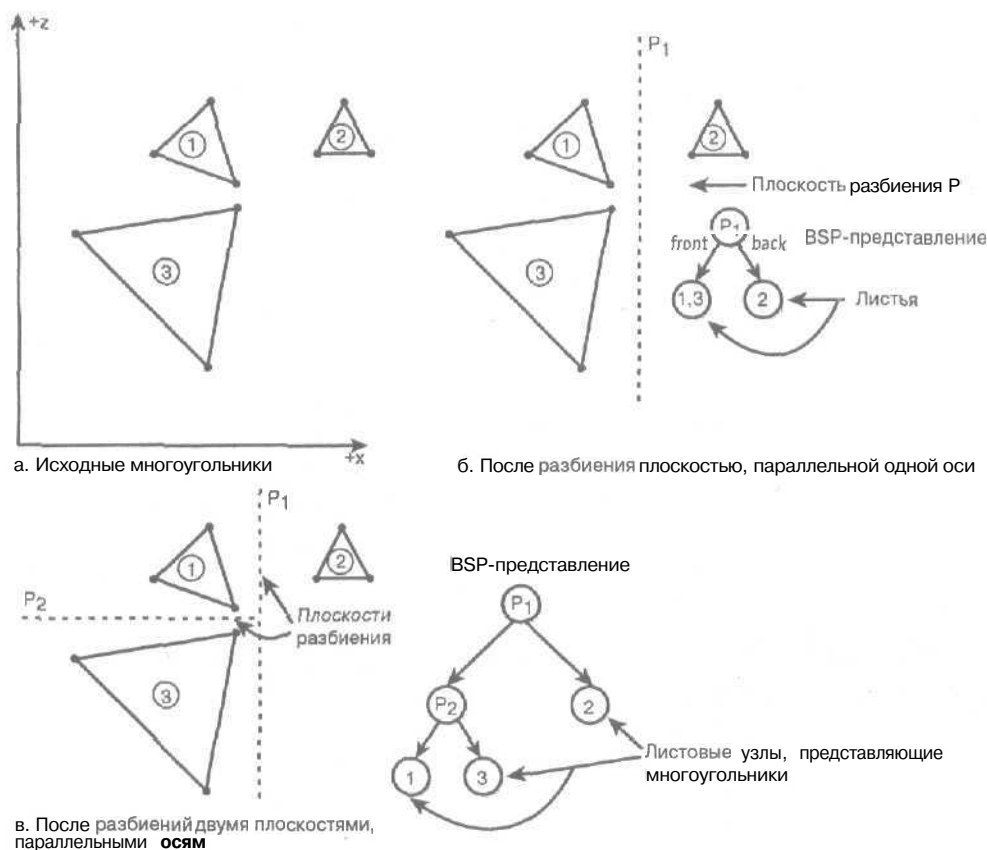


Рис. 13.8. Разбиение многоугольников параллельными осям **плоскостями**

## Разбиение пространства произвольными плоскостями

На рис. 13.9 представлен второй метод разбиения многоугольников: выбираются разделительные плоскости, которые делят пространство на несколько частей. Чтобы найти разделительные плоскости, **позволяющие** равномерно распределить многоугольники и минимизировать число пересечений и т.д., можно воспользоваться некой эвристикой или другим алгоритмом. В результате такого разбиения генерируется **BSP-дерево**, в котором каждый узел определяет подпространство, а каждый конечный лист задает выпуклое подпространство. Таким образом, начиная с корневого узла, правый дочерний узел определяет все, лежащее за разделительной плоскостью, а левый дочерний узел определяет все, находящееся перед разделительной плоскостью. При достижении конечного листового узла дерева оказывается, что он задает выпуклое подпространство, содержащее один или несколько многоугольников, как показано на рис. 13.9.

Далее будет **показано**, как эти разбиения могут помочь нам в решении наших задач, но сначала рассмотрим последний способ разбиения.

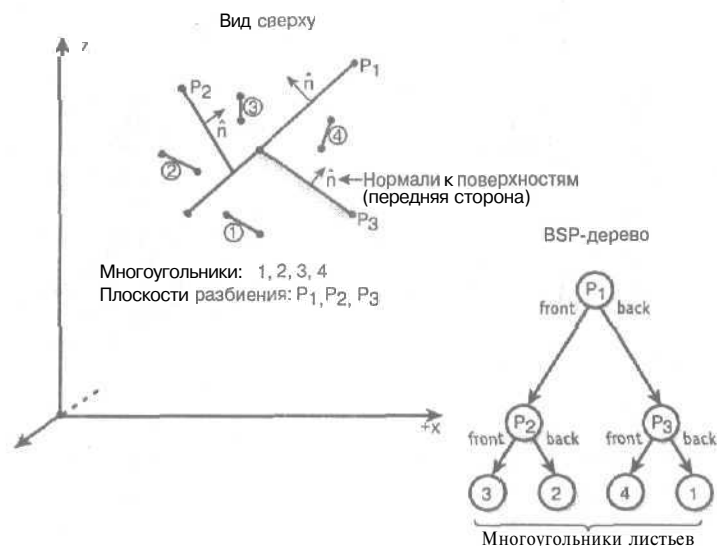


Рис. 13.9. Разбиение многоугольников произвольными плоскостями

## Разбиение с помощью плоскостей, определяемых многоугольниками

Разбиение с помощью определяемых многоугольниками плоскостей осуществляется следующим образом; в качестве разделительной плоскости выбирается некий многоугольник сцены (на данном этапе неважно, какой именно). Плоскость, в которой лежит многоугольник, естественно, бесконечна; мы будем называть ее *гиперплоскостью*. Она делит пространство на два подпространства, как показано на рис. 13.10.

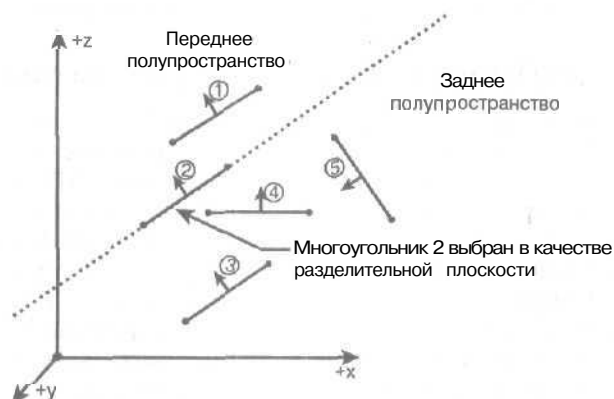


Рис. 13.10. Использование многоугольников в качестве разделительных плоскостей

После этого алгоритм проверяет каждый многоугольник и определяет, с какой стороны разделительной плоскости он находится, лицевой или обратной. Если разделительная плоскость пересекает некий многоугольник(и), данный многоугольник делится точкой (линией) пересечения на два многоугольника, как показано на рис. 13.11.



Рис. 13.11. Разбиение пространства плоскостями многоугольников

НА ЗАМЕТКУ

Термин *гиперплоскость* используется в  $n$ -мерной геометрии применительно к объекту, имеющему на одно измерение меньше, чем само пространство. Например, в трехмерном пространстве гиперплоскостью будет двумерный лист или плоскость. В  $n$ -мерном пространстве гиперплоскость будет иметь размерность  $n-1$ .

После завершения первого шага алгоритма списки многоугольников, находящихся в переднем и заднем полупространствах, полученные в результате выполнения этого шага (списки многоугольников, находящихся впереди и позади разделительной плоскости) рекурсивно обрабатываются аналогичным образом. Эта процедура продолжается до тех пор, пока не будут обработаны все многоугольники и построено **BSP-дерево**. Таким образом, исходный связанный список многоугольников, присоединенный к узлу BSP, используется системой для осуществления разбиения.

В качестве примера рассмотрим рис. 13,12, на котором показано создание двух аналогичных **BSP-деревьев** для состоящей из пяти многоугольников сцены, когда в качестве начальной разделительной плоскости выбираются две различные плоскости. Первый шаг алгоритма состоит в создании одного узла, содержащего разделительную плоскость. Дочерними узлами данного узла становятся два связанных списка: один список представляет многоугольники, находящиеся перед рассматриваемой плоскостью, а другой — многоугольники, расположенные позади нее. Затем каждый из этих списков обрабатывается аналогично: в лицевом списке выбирается разделительная плоскость (пока что она может быть любой) и производится разбиение многоугольников этого списка. Процесс продолжается до тех пор, пока будут получены деревья, имеющие вид, показанный на рис. 13.12.

Созданное **BSP-дерево** имеет ряд особых свойств, которые можно использовать для визуализации, определения столкновений и крупномасштабной отбраковки. Например, можно использовать модифицированный рекурсивный алгоритм симметричного обхода дерева, чтобы для любой точки наблюдения обходить многоугольники строго в порядке от дальних к ближним. Подчеркиваю, для *любой точки наблюдения* в трехмерном пространстве за линейно зависящее от количества объектов время можно определить точный порядок визуализации многоугольников от дальних к ближним (как в случае алгоритма художника), причем это можно сделать, не прибегая к сортировке многоугольников и Z-буферизации.

Кроме того, время поиска для *посещения* каждого узла в *двоичном дереве поиска* (binary search tree, BST) всегда линейно зависит от числа узлов ( $O(n)$ ), поскольку эта задача может быть сведена к задаче поиска объекта в последовательном стеке. Время поиска определенного ключа составляет  $O(\log n)$ .

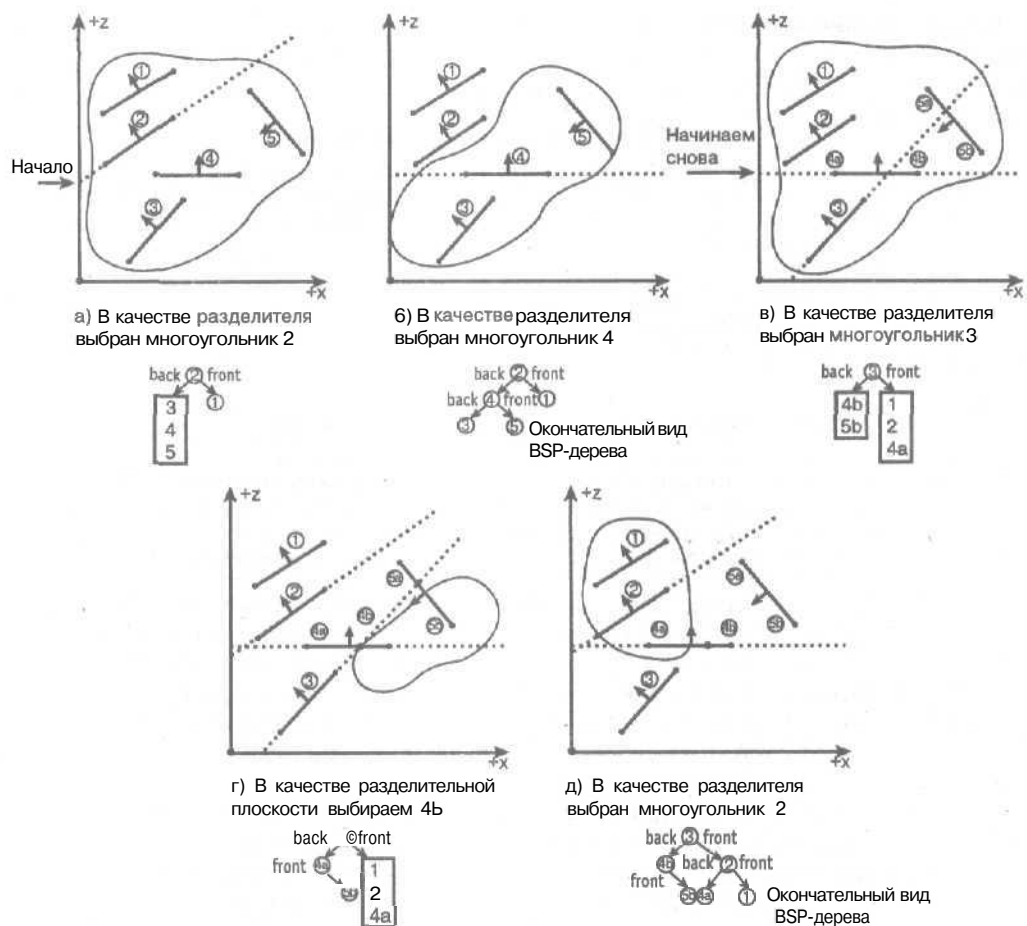


Рис. 13.12. Поэтапное создание BSP-дерева

#### НА ЗАМЕТКУ

Выбор разделительной плоскости имеет очень большое значение, поскольку неудачный выбор может привести к большому количеству расщеплений многоугольников и/или дерево окажется несбалансированным (первая неприятность более существенна). Этот важный аспект алгоритма мы рассмотрим позже. В большинстве случаев можно воспользоваться простой эвристикой, при которой на каждой итерации в качестве возможных разделителей случайным образом отбирается 0,1-1% соответствующего множества многоугольников. Для каждого из них выполняется определенный алгоритм, по результатам которого выбирается многоугольник, который будет выступать в роли **разделителя**, а в роли параметров функции оптимизации выступают количество расщеплений и сбалансированность дерева, и алгоритм продолжает свою работу. Таким образом, если изображение состоит из 10000 многоугольников, на каждой итерации нужно будет проверить не более 10-100 многоугольников, чтобы получить результаты, достаточно близкие к оптимальным.

Итак, теперь вы имеете **общее** представление о том, как создается BSP-дерево. Опишем процедуру более подробно и перечислим этапы, необходимые для составления программного кода. Предположим, что **подлежащий** разбиению связанный спи-

сок многоугольников, составляющих пространство игры или некоего ее уровня, уже создан (с помощью некоего программного средства или функции) и начало этого списка находится в переменной root. Алгоритм создания BSP-дерева выглядит следующим образом.

1. С помощью некой эвристической процедуры выбрать многоугольник из списка и использовать его в качестве разделительной плоскости. Если в списке больше нет многоугольников — выход из алгоритма.
2. Создать два дочерних списка, на один из них указывает указатель front, а на другой — указатель back BSP-узла. В списках содержатся многоугольники, расположенные впереди и позади разделительной плоскости, как показано на рис. 13.13.
3. Рекурсивно обработать список front.
4. Рекурсивно обработать список back.

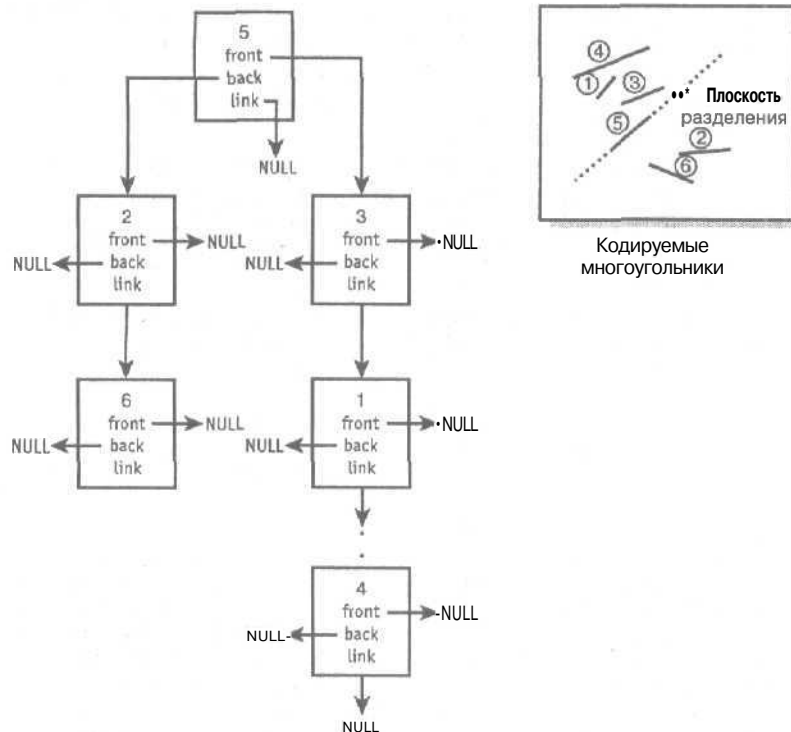


Рис. 13.13. Подробное представление данных BSP и соответствующая структура данных

Конечно же, необходимо рассмотреть множество деталей. Например, как определить, с какой стороны разделительной плоскости находится многоугольник? Какую плоскость следует выбрать в качестве разделительной на каждом этапе (об этом уже упоминалось)? Какие структуры данных лучше всего использовать?

До сих пор мы говорили о BSP-деревьях вообще, однако теперь придется перейти к их конкретному виду. Мы в основном займемся реализацией двумерного алгоритма BSP, который, конечно, будет использоваться в полномасштабном трехмерном игровом процессоре. Однако у меня просто нет времени на создание **полномасштабного** инструментария трехмерного моделирования для **генерирования** трехмерных BSP, а вычисления пересечений произвольных плоскостей достаточно трудоемки и не имеют ничего общего с идеей BSP-деревьев. Поэтому я не буду усложнять ситуацию и рассмотрю создание BSP-миров, состоящих из отрезков прямых, которые будут преобразованы в трехмерные стены. Однако все этапы моделирования и создания BSP изначально будут выполняться в двумерном пространстве, чтобы упростить задачу. Обобщение соответствующей математики для трехмерного случая потребует только дополнительной математической подготовки.

## Отображение/посещение каждого узла BSP-дерева

Как уже упоминалось, BSP-деревья могут использоваться для различных целей: визуализации, определения столкновений, отбраковки и т.п. Однако чаще всего они используются для определения порядка визуализации некоторого уровня игры от дальних объектов к ближним или наоборот для произвольной точки наблюдения. Рассмотрим, как это делается.

К счастью, отображение BSP-дерева гораздо **проще**, чем его построение. Необходимо только написать алгоритм, использующий модифицированный алгоритм симметричного обхода двоичного дерева, который работает следующим образом. Начнем с корневой вершины BSP, как показано на рис. 13.14. Затем выполняется проверка того, с какой стороны от плоскости корневого многоугольника находится точка обзора. Если она находится перед корневым многоугольником, то алгоритм рекурсивно обходит заднюю ветвь, затем данный многоугольник (отображая или обрабатывая его), а затем рекурсивно обходит переднюю ветвь.

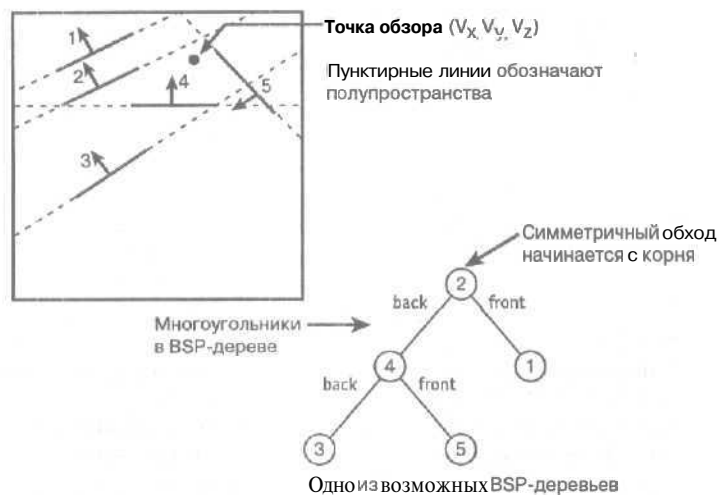


Рис. 13.14. Пример многоугольников и обхода BSP-дерева

Если же точка наблюдения находится сзади плоскости раздела, то выполняются противоположные действия: сначала рекурсивно просматривается передняя ветвь, затем многоугольник и, наконец, задняя ветвь.

Такой порядок действий может показаться несколько запутанным, поэтому, чтобы лучше понять логику происходящего, рассмотрим **еще** один пример. Главным свойством **BSP-дерева** является то, что где бы мы не поместили точку наблюдения после его **создания**, подпространство, в котором эта точка находится, обязательно содержит в себе самый ближний многоугольник и является выпуклым. Таким образом, подпространство, **находящееся** позади указанного многоугольника, гарантированно расположено дальше от точки наблюдения. Следовательно, можно воспользоваться этой способностью **BSP-деревьев** для определения порядка визуализации многоугольников от дальних к **ближним**. На рис. 13.15 показана точка наблюдения  $P_0$  (в мировых координатах), а также многоугольники, составляющие **BSP**.

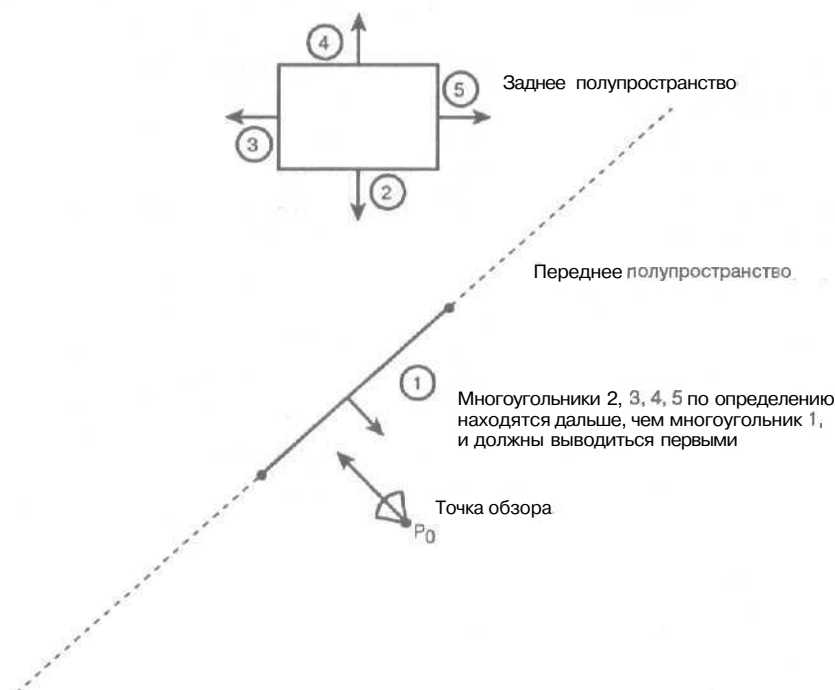


Рис. 13.15. Многоугольники текущего подпространства находятся ближе к точке наблюдения

Прежде всего отметим, что точка наблюдения находится перед многоугольником 1. Следовательно, она расположена ближе всего к этому многоугольнику, поскольку, если мы находимся с передней стороны и других разделительных плоскостей нет, то, по определению, мы находимся в выпуклом подпространстве и ближайшим многоугольником является многоугольник 1. Из этого вытекает, что многоугольники, расположенные позади многоугольника 1, находятся дальше от нас, и, посещая эти многоугольники, можно рекурсивно применять те же рассуждения на любом этапе. Таким образом, если мы посещаем данные многоугольники (продолжая рекурсивный поиск до тех пор, пока не достигнем наиболее удаленного многоугольника), затем посещаем каждый узел (рисует его), то в результате получается порядок визуализации от дальних многоугольников к ближним. Соответственно, если при посещении узлов используется противоположная стратегия поиска, результатом будет порядок визуализации от ближних к дальним.

Вернемся к **BSP-дереву** на рис. 13.14 и предположим, что точка наблюдения расположена так, как показано на этом рисунке. Применим алгоритм **шаг** за шагом и посмотрим, правильно ли он работает.

1. Начнем с многоугольника 2. Точка наблюдения находится в заднем полупространстве; следовательно, начнем с обхода переднего подпространства многоугольника 2.
2. Поскольку многоугольник 1 не имеет дочерних многоугольников, он **визуализируется**, снимается со стека и наверху вновь оказывается многоугольник 2.
3. Многоугольник 2 визуализируется, и выполняется обход его заднего полупространства.
4. Точка наблюдения находится в переднем полупространстве многоугольника 4; следовательно, начинаем с заднего полупространства многоугольника 4.
5. Поскольку многоугольник 3 не имеет дочерних многоугольников, он визуализируется, снимается со стека и наверху вновь оказывается **многоугольник 4**.
6. Визуализируется многоугольник 4.
7. Исследуется переднее полупространство многоугольника 4.
8. Поскольку многоугольник 5 не имеет дочерних многоугольников, он **визуализируется** и снимается со стека, после чего стек оказывается пуст.

Таким образом, многоугольники рисуются в последовательности 1, 2, **3, 4, 5**, т.е. в порядке от дальних к ближним.

Алгоритм **посещения** BSP будет работать независимо от того, какие многоугольники были выбраны в качестве разделительных плоскостей в ходе создания BSP-дерева, независимо от точки наблюдения и ориентации. В заключение приведем алгоритм обхода, записанный с **помощью** псевдокода C/C++.

Предположим, что каждый узел имеет передний и задний указатели.

```
void Bsp_Traverse(BSPNODE* root, VECTOR4D viewpoint)
{
    if (root==NULL) return;

    if (точка наблюдения находится с передней стороны корня)
    {
        // Выполняется прямой симметричный обход
        Bsp_Traverse(root->back, viewpoint);
        Visit_Polygon(root->poly);
        Bsp_Traverse(root->front, viewpoint);
    }
    else
    {
        // Выполняется симметричный обход в обратном порядке
        Bsp_Traverse(root->front, viewpoint);
        Display_Polygon(root->poly);
        Bsp_Traverse(root->back, viewpoint);
    } // else
} // Bsp_Traverse
```

Не правда ли, все просто? Конечно, для определения, впереди или позади многоугольника находится точка наблюдения, необходимо вычисление соответствующего скалярного произведения. Для простоты это вычисление в данном фрагменте не показано, чтобы показать красоту алгоритма более наглядно.

Теперь необходимо разобраться, какие структуры данных и функции необходимы, чтобы создать работающую демонстрационную версию BSP и встроить ее в наш **существующий** трехмерный игровой процессор с поддержкой освещения, текстуры, отсечения и т.п.

## Функции и структуры данных BSP-деревьев

Реализация **BSP-дерева** и поддерживающих его функций не так проста, как может показаться, она требует принятия ряда важных решений. Даже если нас интересует только демонстрация метода, создание такой демонстрационной версии — нелегкая задача. Поэтому я старался сделать все как можно проще, чтобы были понятны основные идеи. Начнем со структуры данных, используемой для представления отдельной стены в виде **BSP-узла**.

В **настоящий** момент наш трехмерный игровой процессор полностью основан на **треугольниках**, а не на четырехугольниках. Это проектное решение было принято для того, чтобы упростить все элементы представления, освещения, отсечения и т.д. Однако треугольники — не лучший способ представления при моделировании внутренних помещений, поскольку большинство интерьеров состоит из ровных плоских поверхностей, главным образом, четырехугольных. Чтобы не использовать два треугольника для представления четырехугольника, было принято решение просто создать новый тип многоугольника с четырьмя вершинами вместо трех и выполнять все операции по созданию BSP с этим новым представлением. Затем, в процессе обхода BSP можно преобразовать каждый многоугольник, являющийся узлом BSP, в два треугольника, как показано на рис. 13.16.

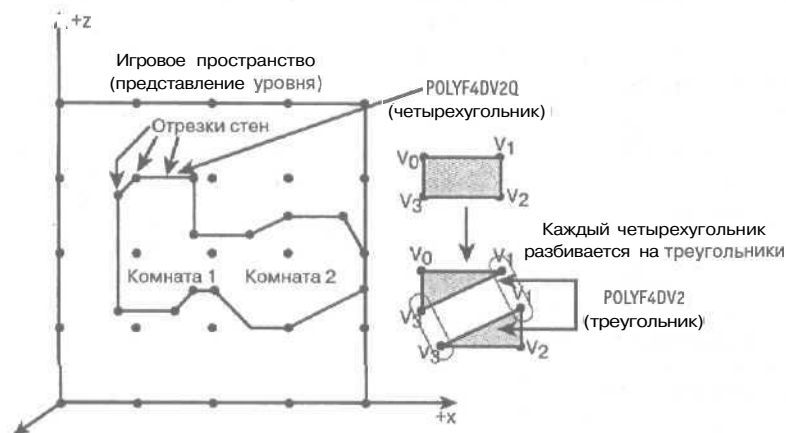


Рис. 13.16. Использование представления, основанного на четырехугольниках, для упрощения создания BSP-дерева

С другой стороны, хотелось бы использовать уже знакомые структуры. Поэтому давайте возьмем тип **POLYF4DV2** и добавим к нему еще одну вершину, а к имени — букву **Q**.

```
// Самодостаточный четырехугольник, используемый в списке
// визуализации, версия 2. Нужен для представления стен,
// поскольку стены - четырехугольники
typedef struct POLYF4DV2Q_TYP
{
    int state;    // Информация о состоянии
    int attr;    // Физические атрибуты многоугольника
}
```

```

int color;          // Цвет многоугольника
int lit_color[4];   // Хранит цвета после освещения, 0 —
                    // для плоского затенения, 0,1,2 - для
                    // цветов вершины после освещения
BITMAP_IMAGE_PTR texture; // Указатель на информацию о
                    // текстуре для простого отображения
                    // текстуры

int mati;          // Индекс материала; -1 при отсутствии
                    // материала

float nlength;      // Длина нормали к многоугольнику, если
                    // она не нормализована
VECTOR4D normal;    // Обобщенная нормаль к многоугольнику

float avg_z;        // Среднее значение z вершин,
                    // используемое для простой сортировки

VERTEX4DTV1 vlist[4]; // Вершины четырехугольника
VERTEX4DTV1 tvlist[4]; // Вершины после преобразований

POLYF4DV2Q_TYP *next; // Указатель на следующий
                      // многоугольник в списке
POLYF4DV2Q_TYP *prev; // Указатель на предыдущий
                      // многоугольник в списке

} POLYF4DV2Q; *POLYF4DV2Q_PTR;

```

Элементы, измененные по сравнению с предыдущей структурой, **POLYF4DV2**, выделены полужирным шрифтом. Структуру **POLYF4DV2Q** можно использовать для представления отдельных четырехугольников, но нам необходимо преобразовать ее в **BSP-узел**, который не только содержит один из четырехугольников, представляющий как сам многоугольник, так и (неявно) плоскость раздела, но и поддерживает передний и задний указатели для построения **BSP-структуры**. Назовем эту структуру обобщенным **BSP-узлом**.

```

// Обобщенный bsp-узел. Все bsp-деревья строятся из таких
// структур; корневой узел bsp также является такой
// структурой
typedef struct BSPNODEV1_TYP
{
    int id;          // Идентификатор для отладки
    POLYF4DV2Q wall; // Четырехугольник текущей стены

    struct BSPNODEV1_TYP *link; // Указатель на следующую
                                // стену
    struct BSPNODEV1_TYP *front; // Указатель на стены,
                                // находящиеся впереди
    struct BSPNODEV1_TYP *back;  // Указатель на стены,
                                // находящиеся позади

} BSPNODEV1; *BSPNODEV1_PTR;

```

Структура **BSPNODEV1** будет базовой единицей **BSP-дерева**; ее графическое представление показано на рис. 13.17. Каждый узел, как правило, состоит из четырехугольного много-

угольника, идентификационного номера для отладки и трех указателей, два из которых мы уже обсуждали (передний и задний указатели). Последний указатель называется Link и используется для объединения BSP-узлов в связанный список в ходе построения дерева. Содержащаяся в link информация используется в качестве исходных данных алгоритмом двоичного разбиения пространства. Таким образом, можно создать связанный список стен с помощью любой программы; затем этот список обрабатывается алгоритмом двоичного разбиения пространства, что позволяет преобразовать игровой уровень в BSP.

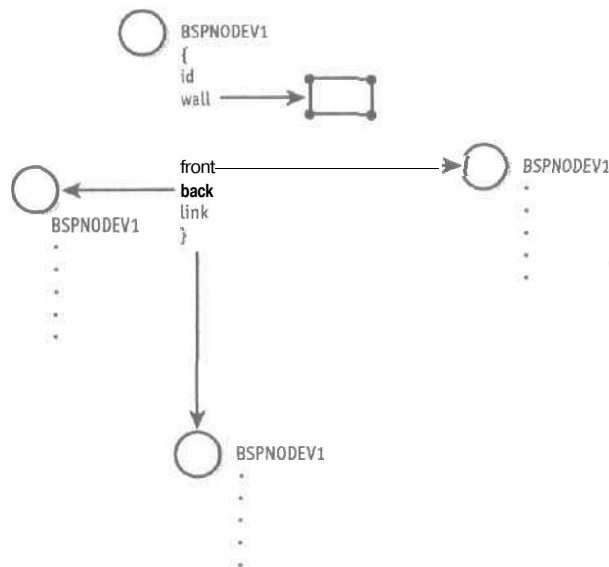


Рис. 13.17. Графическое представление BSPNODE1

Теперь предположим, что некая программа или инструмент позволили пользователю ввести множество стен, и сгенерировали связанный список, в котором для связи стен используются элементы Link, как показано на рис. 13.18. Попробуем написать программу, которая на основании этих данных создает BSP-дерево.

## Создание BSP-дерева

С теоретической точки зрения, построение BSP-дерева — не очень сложная задача, но когда начинается перемещение указателей, распределение памяти, а компланарные стены имеют общие точки, все очень быстро усложняется. Тем не менее, я попытался сделать функцию максимально понятной. В общих чертах данная функция работает следующим образом: она получает список стен, затем первый элемент списка используется в качестве разделяющей плоскости, а стены, находящиеся перед данной плоскостью и позади нее, помещаются в два списка, на которые указывают соответственно передний и задний указатели первого узла разбиения. Отдельные списки связываются при помощи поля link. Каждый из полученных списков затем используется в качестве входной информации в последующих рекурсивных вызовах функции создания BSP. Этот процесс продолжается до тех пор, пока все передние и задние списки будут содержать только по одному многоугольнику или не будут содержать ни одного. На этом построение BSP-дерева завершается.

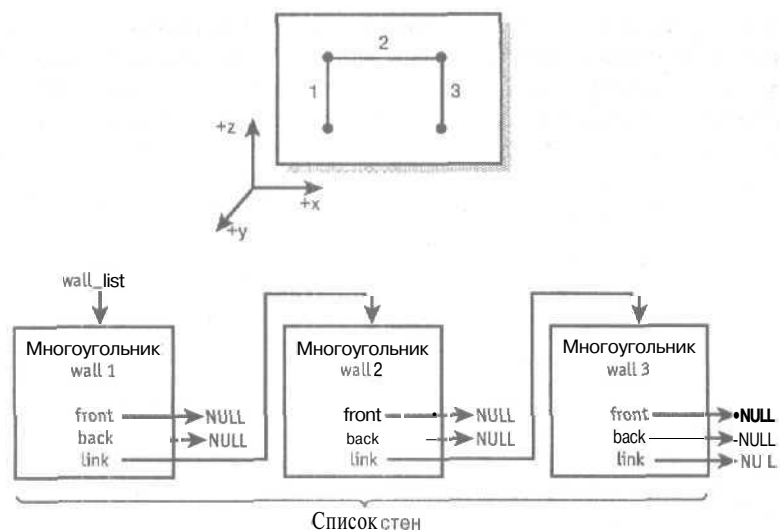


Рис. 13.18. Список стен перед преобразованием в BSP-дерево

В описанном процессе **существуют** определенные сложности и требуется рассмотреть некоторые частные случаи. Первая сложность заключается в том, как определить, с какой стороны разделяющей плоскости находится многоугольник. На рис. 13.19 показаны три возможных случая.

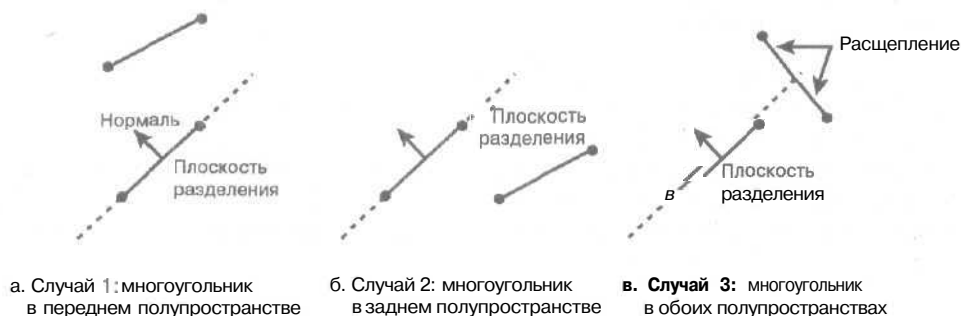


Рис. 13.19. Варианты возможной геометрической ориентации стен

1. Оба конца отрезка расположены с лицевой стороны рассматриваемой разделяющей плоскости.
2. Оба **конца** отрезка находятся с обратной стороны разделяющей плоскости.
3. Один конец отрезка находится с лицевой стороны, а второй — с обратной стороны разделяющей плоскости; такой многоугольник необходимо разбить на два.

Случаи 1 и 2 легко проверяются и достаточно просто обрабатываются. Чтобы проверить, с какой стороны находится каждая вершина многоугольника относительно проверяемой стены, вычисляется скалярное произведение нормали к плоскости раздела и вектора к каждой из проверяемых вершин многоугольника. Стены, **находящиеся** с лицевой стороны, добавляются в передний список разделяющей стены, а стены, **находящиеся** с обратной стороны, добавляются к заднему связанному списку. Однако в случае 3 возникают определенные сложности.

В этом случае необходимо продлить разделительную плоскость и определить, в каком месте она пересекает рассматриваемую стену. Это легко сделать, поскольку стены представлены двумерными отрезками в плоскости  $xz$ . Вычисление точки пересечения двух стен превращается в задачу определения точки пересечения двух прямых, как показано на рис. 13.20.

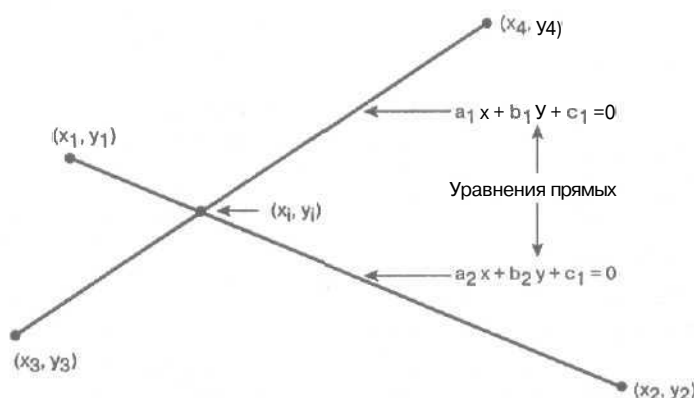


Рис. 13.20. Определение точки пересечения двух прямых

Существует множество способов вычисления точки пересечения двух бесконечных прямых, но мы будем использовать простой метод, основанный на двух линейных уравнениях, а не параметрический метод, поскольку это потребовало бы дополнительной работы по преобразованию линейных отрезков в параметрическую форму и в данном случае не принесло бы никаких выгод. В реальном BSP-процессоре могут использоваться параметрические прямые, но в нашем случае проще воспользоваться линейными уравнениями. Кроме того, напомним, что затраченное на создание BSP-дерева время не является критическим (критическим является только время его отображения), поскольку после того, как дерево создано, его можно просто загружать с диска и использовать. Таким образом, лишние вычисления в данном случае не ведут к нежелательным последствиям.

Обратимся к рис. 13.20 и найдем точку пересечения. Имеется два линейных уравнения вида

$$a_1x + b_1y = c_1,$$

$$a_2x + b_2y = c_2,$$

каждое из которых представляет одну из рассматриваемых прямых. Данную систему уравнений можно решить с помощью правила Крамера, умножения матриц или с помощью метода подстановки, (см. главы 4, "Запутанный мир математики", и 5, "Создание математической библиотеки"). Воспользуемся правилом Крамера, согласно которому решения системы линейных уравнений можно вычислить, если заменить в матрице коэффициентов столбец коэффициентов при искомой переменной столбцом вектора свободных членов и найти определитель полученной матрицы. Результат этих вычислений затем делится на определитель исходной матрицы коэффициентов. Эта процедура выполняется для каждой неизвестной переменной. Матрица коэффициентов нашей системы имеет вид

$$C = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}, \text{ а ее определитель равен } \det(C) = (a_1b_2 - a_2b_1).$$

Найти  $x$  можно путем следующих вычислений:

$$x_i = \frac{\det \begin{pmatrix} c_1 & b_1 \\ c_2 & b_2 \end{pmatrix}}{\det(C)} = \frac{(b_2 c_1 - b_1 c_2)}{(a_1 b_2 - a_2 b_1)},$$

а значение  $y$  находится по следующей формуле:

$$y_i = \frac{\det \begin{pmatrix} a_1 & c_1 \\ a_2 & c_2 \end{pmatrix}}{\det(C)} = \frac{(a_1 c_2 - a_2 c_1)}{(a_1 b_2 - a_2 b_1)}.$$

Единственная проблема состоит в том, как записать уравнения прямых в виде линейных уравнений. Для этого можно использовать уравнение прямой в форме

$$m(x - x_0) = (y - y_0),$$

где  $m$  — коэффициент наклона, равный  $(y_1 - y_0)/(x_1 - x_0)$ . Преобразуя данное уравнение к линейному виду, получим:

$$(m)x + (-1)y = (mx_0 - y_0),$$

что соответствует виду  $ax + by = c$  (где  $a = m$ ,  $b = -1$ , и  $c = (mx_0 - y_0)$ ).

Исходя из приведенных выше рассуждений (которые должны быть вам знакомы, поскольку мы рассматривали этот материал в главах 4, "Запутанный мир математики", и 5, "Создание математической библиотеки"), **следующая** функция вычисляет координаты пересечения двух прямых по заданным координатам концов отрезков.

```
void Intersect_Lines(float x0, float y0, float x1, float y1,
                    float x2, float y2, float x3, float y3,
                    float *xi, float *yi)
{
    // Данная функция вычисляет, где пересекаются заданные
    // прямые, и возвращает точку пересечения (считая, что
    // эти прямые пересекаются). Функция может обрабатывать
    // как вертикальные, так и горизонтальные прямые. Данная
    // функция просто использует математические выражения,
    // но поскольку эти вычисления проводятся до самой игры,
    // нам нет необходимости ускорять процесс. При желании
    // можно использовать параметрические прямые, но в
    // данном случае проще находить пересечение 2-х
    // бесконечных прямых, а не отрезков

    float a1, b1, c1, // коэффициенты линейных уравнений
          a2, b2, c2,
    det_inv, // Число, обратное определителю матрицы
            // коэффициентов
    m1, m2; // Угловые коэффициенты прямых

    // Вычисление угловых коэффициентов; в случае
    // бесконечности используется просто очень большое число
    if ((x1 - x0) != 0)
        m1 = (y1 - y0) / (x1 - x0);
    else
        m1 = (float)1.0E+20; // Достаточно большое число

    if ((x3 - x2) != 0)
        m2 = (y3 - y2) / (x3 - x2);
    else
```

```

    m2 - (float)1.0E+20; // Достаточно большое число

// Вычисление констант
a1 = m1;
a2 = m2;

b1 = -1;
b2 = -1;

c1 = (y0-m1*x0);
c2 = (y2-m2*x2);

// Вычисление обратного определителя
det_inv = 1 / (a1*b2 - a2*b1);

// Вычисление xi и yi по правилу Крамера
*xi = ((b1*c2 - b2*c1)*det_inv);
*yi = ((a2*c1 - a1*c2)*det_inv);

} // Intersect_Lines

```

Данная функция получает в качестве параметров значения координат  $x$  и  $y$  конечных точек двух отрезков и пару указателей, задающих, куда возвращать точку пересечения,

#### НА ЗАМЕТКУ

В данной функции предполагается, что прямые не являются параллельными, т.е. что они пересекаются. Если же прямые параллельны, это приведет к делению на ноль, поскольку определитель для двух параллельных линий всегда равен нулю. Этой проблемы можно избежать, если первым делом проверить, являются ли линии параллельными (в рамках заданного допуска), и в случае положительного ответа предусмотреть код, сигнализирующий о наличии проблемы.

Вызов функции происходит только в том случае, если линии не параллельны. Читателям в качестве упражнения предлагается самостоятельно написать код проверки условия параллельности прямых.

Выяснив, как находить точку пересечения разделительной плоскости с проверяемой стеной, необходимо разделить данную стену на две стены в точке пересечения  $(x_i, y_i)$  (в трехмерном пространстве  $(x_i, z_i)$  в плоскости  $xz$ ). Одна часть расщепленной стены присоединяется к переднему связанному списку, а вторая часть включается в задний связанный список узла, представляющего разделительную плоскость.

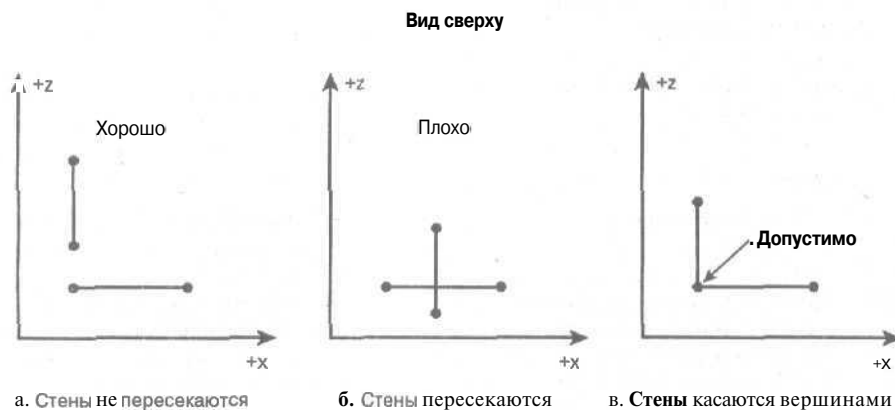
## Стратегии расщепления

Прежде чем двигаться дальше, поговорим об оптимизации. Когда стена расщепляется на две стены, вместо расщепления можно добавить эту стену в неизменном виде к обоим поддеревьям (и переднему, и заднему), добавив поле, в котором для дальнейшей работы указана точка пересечения. Таким образом можно избежать разбиения многоугольников, хотя при этом у нас фактически получится две копии многоугольника в двух списках.

Может показаться, что такое решение приведет к возникновению проблем, однако это не так. При обходе **BSP-дерева** любой посещенный многоугольник помечается с помощью дескриптора или некоторым иным образом. Если многоугольник имеет две связи, то при первом посещении будет установлен флаг и все другие посещения будут игнорироваться. Кроме того, при использовании данного метода многоугольник может расщепляться несколько раз и иметь несколько точек расщепления. Поэтому нужна стратегия, позволяю-

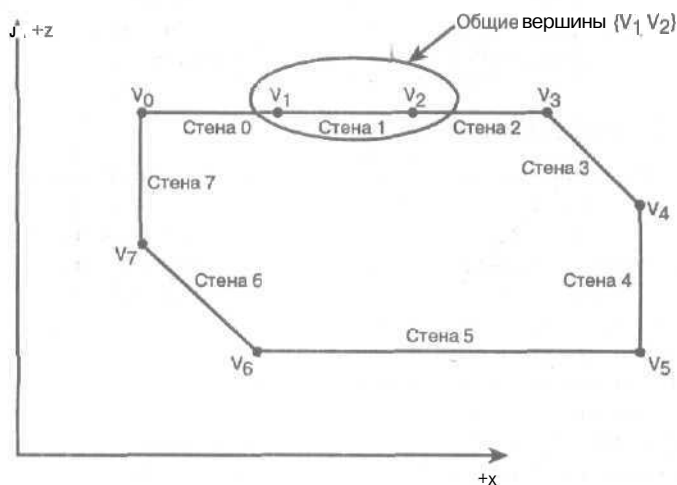
щая объединять многочисленные точки **расщепления** в массив или аналогичную структуру. Тем не менее, алгоритм меток работает и в этом случае: после первого посещения многоугольника все последующие посещения будут игнорироваться. Это обеспечивает строгий порядок поиска по BSP от дальних многоугольников к ближним или от ближних к дальним. Однако применять этот метод здесь мы не будем, чтобы не усложнять себе жизнь.

Вернемся вновь к построению BSP-дерева и обсудим различные частные случаи. Мы считаем, что никакие две стены не пересекаются, как показано на рис. 13.21. Это не сильно ограничивает общность, поскольку если необходимо сделать так, чтобы стены пересекались, всегда можно использовать две стены, по одной с каждой стороны секущей плоскости.



*Рис. 13.21. Стены не должны пересекаться*

Необходимо рассмотреть еще один часто встречающийся случай, когда две стены имеют **общую** вершину. На рис. 13.22 изображено несколько компланарных отрезков стен. Проблема в том, что в ходе наших вычислений общие точки нельзя классифицировать как находящиеся перед разделительной плоскостью или позади нее. Как же поступить в данном случае?



*Рис. 13.22. Компланарные многоугольники с общими вершинами*

Чтобы ответить на данный вопрос, спросим себя, что означает ситуация, когда две стены имеют **общую** сторону? Когда две стены имеют общую вертикальную сторону (что при виде сверху выглядит как общая точка), то эта точка находится ни "перед", ни "за", а в точности *на* самой разделительной плоскости; следовательно, чтобы решить, как расположена рассматриваемая стена по отношению к разделительной плоскости, необходимо проанализировать *вторую* конечную точку стены.

Если оказалось, что разделительная плоскость и рассматриваемая стена имеют общую сторону или точку (в зависимости от того, трехмерный или двумерный случай анализируется), для решения вопроса, с какой стороны разделительной плоскости находится данный многоугольник, используется вторая конечная вершина стены. Возникает еще один вопрос: что будет, если вторая конечная точка также принадлежит разделительной плоскости? Ничего страшного: в таком случае не имеет значения, куда отнести рассматриваемый **многоугольник**, в передний или задний список.

При наличии компланарных стен можно произвести дополнительную оптимизацию, объединив эти стены. Например, зачем выполнять отдельные вычисления для каждого отрезка на рис. 13.22, если известно (или следует из вычислений), что все они компланарные? В таком случае для представления всех таких отрезков можно использовать единый отрезок или стену, и тогда не придется отдельно рассматривать случаи компланарности сегментов стен.

Конечно же, все это приводит к определенным усложнениям. Становится **более** сложной структура данных для хранения BSP: хотя можно считать коллекцию сегментов стен единой стеной или плоскостью, нужно иметь возможность разбить их другими разделительными плоскостями. Поэтому обычно формируется связанный список компланарных стен, выполняются вычисления пересечений так, как будто они образуют единую стену, а затем производится разбиение этой стены на множество стен, которые прикрепляются к переднему и заднему спискам плоскости разделения на каждой итерации алгоритма (рис. 13.23). Хотя это весьма трудоемкий процесс, нам придется проделать **соответствующую** работу по написанию кода, чтобы создать игровой процессор.

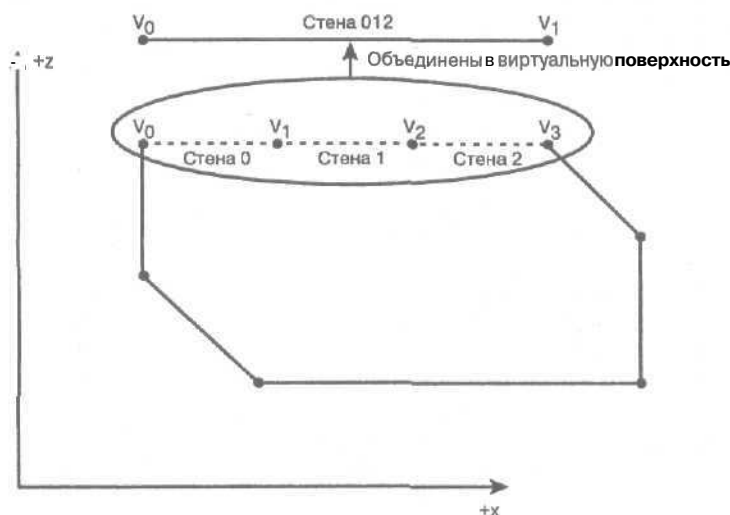


Рис. 13.23. Объединение компланарных стен в единую поверхность для упрощения вычислений

Теперь, после того, как мы описали **реализацию** алгоритма, рассмотрим, как же выглядит функция, **создающая** BSP-дерево по связанному списку стен, сформированному некоторой другой программой или функцией.

```
void Bsp_Build_Tree(BSPNODEV1_PTR root)
```

```
!
// Функция рекурсивно строит bsp-дерево, начиная от
// корня списка стен. Функция получает неупорядоченный
// связанный список стен, использует стену, находящуюся
// в голове списка, в качестве разделительной плоскости
// и производит разбиение пространства, вычисляя, с
// какой стороны разделительной плоскости находятся
// остальные стены. Стены, находящиеся с лицевой и
// обратной сторон образуют два новых связанных списка,
// каждый из которых обрабатывается аналогично. По
// окончании процесса будет построено полное bsp-дерево,
// в котором каждой стене соответствует в точности один
// узел/лист
```

```
static BSPNODEV1_PTR
```

```
    next_wall,    // Указатель на следующую
                  // обрабатываемую стену
    front_wall,   // Лицевая стена
    back_wall,    // Обратная стена
    temp_wall;    // Временная стена
```

```
static float
```

```
    dot_wall_1,      // Скалярное произведение для
    dot_wall_2,      // тестируемой стены
    wall_x0,wall_y0,wall_z0, // Рабочие переменные
    wall_x1,wall_y1,wall_z1, // для тестируемой стены
    pp_x0,pp_y0,pp_z0,   // рабочие переменные для
    pp_x1,pp_y1,pp_z1,   // разделительной плоскости
    xi,zi;             // Точки пересечения для
                        // случая, когда
                        // разделительная плоскость
                        // делит стену
```

```
static VECTOR4D
```

```
    test_vector_1,    // Тестовые векторы, идущие от
    test_vector_2;    // плоскости разделения к тестируемой
                        // стене для проверки, с какой
                        // стороны плоскости находится стена
```

```
static int front_flag - 0,    // Флаги, указывающие, что
    back_flag - 0,           // стена находится с лицевой
                              // или обратной стороны
                              // плоскости разделения
    index;                   // Индекс цикла
```

```
// Раздел 1 //////////////////////////////////////
```

```
// Проверка, является ли данное дерево законченным
```

```

if(root==NULL)
    return;

// Выбираем в качестве разделительной плоскости корневую
// вершину, производим разбиение многоугольников с ее
// использованием
next_wall = root->link;
root->link = NULL;

// Извлечение двух верхних вершин образующей плоскость
// разделения стены для упрощения вычислений
pp_x0 = root->wall.vlist[0].x;
pp_y0 = root->wall.vlist[0].y;
pp_z0 = root->wall.vlist[0].z;

pp_x1 = root->wall.vlist[1].x;
pp_y1 = root->wall.vlist[1].y;
pp_z1 = root->wall.vlist[1].z;

// Раздел 2 //////////////////////////////////////

// Проверка, все ли стены подвергнуты разбиению
while(next_wall)
{
    // Проверка, по какую сторону от разделительной
    // плоскости, заданной корневым многоугольником
    // списка, находится тестируемая стена

    // Сначала вычисляются вектора от точки на
    // разделительной плоскости к точкам тестируемой
    // стены
    VECTOR4D_Build(&root->wall.vlist[0].v,
        &next_wall->wall.vlist[0].v,
        &test_vector_1);

    VECTOR4D_Build(&root->wall.vlist[0].v,
        &next_wall->wall.vlist[1].v,
        &test_vector_2);

    // Затем находятся скалярные произведения каждого
    // вектора тестирования и нормали к поверхности и
    // анализируются их знаки, чтобы определить
    // полупространство
    dot_wall_1 =
        VECTOR4D_Dot(&test_vector_1,&root->wall.normal);
    dot_wall_2 =
        VECTOR4D_Dot(&test_vector_2,&root->wall.normal);

    // Раздел 3 //////////////////////////////////////

    // Выполнение тестов

    // Случай 0, плоскость разделения и тестируемая

```

```
// стена имеют общую точку. Это частный случай,
// который необходимо рассмотреть. Чтобы сократить
// код, установим пару флагов, и тогда само
// включение стены в BSP будет обрабатывать
// следующий случай
```

```
// Сброс флагов
front_flag - back_flag = 0;
```

```
// Определение, является ли стена касательной к
// разделительной стене
if (VECTOR4D_Equal(&root->wall.vlist[0].v,
    &next_wall->wall.vlist[0].v))
{
    // Точка p0 разделительной плоскости та же, что
    // и p0 тестируемой стены. Нужно только
    // выяснить, с какой стороны лежит точка
    // тестируемой стены p1
    if (dot_wall_2 > 0)
        front_flag = 1;
    else
        back_flag = 1;
} // if
else if (VECTOR4D_Equal(&root->wall.vlist[0].v,
    &next_wall->wall.vlist[1].v))
{
    // Точка p0 разделительной плоскости та же, что
    // и p1 тестируемой стены. Нужно только
    // выяснить, с какой стороны лежит точка
    // тестируемой стены p0
    if (dot_wall_1 > 0)
        front_flag = 1;
    else
        back_flag = 1;
} // if
else if (VECTOR4D_Equal(&root->wall.vlist[1].v,
    &next_wall->wall.vlist[0].v))
{
    // Точка p1 разделительной плоскости та же, что
    // и p0 тестируемой стены. Нужно только
    // выяснить, с какой стороны лежит точка
    // тестируемой стены p1
    if (dot_wall_2 > 0)
        front_flag = 1;
    else
        back_flag = 1;
} // if
else if (VECTOR4D_Equal(&root->wall.vlist[1].v,
    &next_wall->wall.vlist[1].v))
{

```

```

// Точка p1 разделительной плоскости та же, что
// и p1 тестируемой стены. Нужно только
// выяснить, с какой стороны лежит точка
// тестируемой стены pO
if (dot_wall_1 > 0)
    front_flag = 1;
else
    back_flag = 1;

} // if

// Раздел 4 //////////////////////////////////////

// Случай 1: оба знака одинаковы или был установлен
// лицевой или обратный флаг
if ((dot_wall_1 >= 0 && dot_wall_2 >= 0) || front_flag)
{
    // Поместить данную стену в передний список
    if (root->front == NULL)
    {
        // Это первый узел
        root->front = next_wall;
        next_wall = next_wall->link;
        front_wall = root->front;
        front_wall->link = NULL;
    } // if
    else
    {
        // Это n-й узел
        front_wall->link = next_wall;
        next_wall = next_wall->link;
        front_wall = front_wall->link;
        front_wall->link = NULL;
    } // else
} // if оба положительны

// Раздел 5 //////////////////////////////////////

// Подпункт для обратной стороны
elseif ((dot_wall_1 < 0 && dot_wall_2 < 0) || back_flag)
{
    // Поместить данную стену в задний список
    if (root->back == NULL)
    {
        // Это первый узел
        root->back = next_wall;
        next_wall = next_wall->link;
        back_wall = root->back;
        back_wall->link = NULL;
    } // if
    else
    {

```

```

        // Это n-й узел
        faack_wall->link=next_wall;
        next_wall = next_wall->link;
        back_wall = back_wall->link;
        back_wall->link = NULL;
    } // else
} // if оба отрицательны

// Случай 2, знаки различны, необходимо произвести
// расщепление стены

// Раздел 6 //////////////////////////////////////

else if ((dot_wall_1 < 0 && dot_wall_2 >= 0) ||
        (dot_wall_1 >= 0 && dot_wall_2 < 0))
{
    // Плоскость разделения делит стену надвое,
    // данную стену необходимо разделить на две
    // стены

    // Извлекаются верхние две вершины тестируемой
    // стены для упрощения вычислений
    wall_x0 = next_wall->wall.vlist[0].x;
    wall_y0 = next_wall->wall.vlist[0].y;
    wall_z0 = next_wall->wall.vlist[0].z;

    wall_x1 = next_wall->wall.vlist[1].x;
    wall_y1 = next_wall->wall.vlist[1].y;
    wall_z1 = next_wall->wall.vlist[1].z;

    // Вычисляется точка пересечения стен. Заметим,
    // что пересечение происходит в плоскости xz
    Intersect_Lines(wall_x0, wall_z0,
                    wall_x1, wall_z1,
                    pp_x0, pp_z0, pp_x1, pp_z1,
                    &xi, &zi);

    // Здесь начинается самая сложная часть, в
    // которой нам нужно разделить стену, создав две
    // новые стены. Мы создадим две новые стены,
    // поместив их, соответственно, в передний и
    // задний списки, а затем удалим исходную стену

    // Обработка первой стены...

    // Выделение памяти для стены
    temp_wall =
        (BSPNODEV1_PTR)malloc(sizeof(BSPNODEV1));

    // Определение связей
    temp_wall->front = NULL;
    temp_wall->back = NULL;
    temp_wall->link = NULL;

```

```

// Нормаль к многоугольнику остается прежней
temp_wall->wall.normal = next_wall->wall.normal;
temp_wall->wall.nlength = next_wall->wall.nlength;

// Цвет многоугольника остается прежним
temp_wall->wall.color = next_wall->wall.color;

// Материал многоугольника остается прежним
temp_wall->wall.mati = next_wall->wall.mati;

// Текстура многоугольника остается прежней
temp_wall->wall.texture = next_wall->wall.texture;

// Атрибуты многоугольника остаются прежними
temp_wall->wall.attr = next_wall->wall.attr;

// Состояния многоугольника остаются прежними
temp_wall->wall.state = next_wall->wall.state;

// Добавление члена, указывающего на расщепление
temp_wall->id = next_wall->id + WALL_SPLIT_ID;

// Вычисление вершин стены
for (index = 0; index < 4; index++)
{
    temp_wall->wall.vlist[index].x =
        next_wall->wall.vlist[index].x;
    temp_wall->wall.vlist[index].y =
        next_wall->wall.vlist[index].y;
    temp_wall->wall.vlist[index].z =
        next_wall->wall.vlist[index].z;
    temp_wall->wall.vlist[index].w = 1;

    // Копирование атрибутов вершин, координат
    // текстуры, нормали
    temp_wall->wall.vlist[index].attr =
        next_wall->wall.vlist[index].attr;
    temp_wall->wall.vlist[index].n =
        next_wall->wall.vlist[index].n;
    temp_wall->wall.vlist[index].t =
        next_wall->wall.vlist[index].t;
} // for index

// Модификация вершин 1 и 2 с учетом точки
// пересечения. Координата y остается прежней,
// поскольку она не зависит от разбиения стены
temp_wall->wall.vlist[1].x = xi;
temp_wall->wall.vlist[1].z = zi;

temp_wall->wall.vlist[2].x = xi;
temp_wall->wall.vlist[2].z = zi;

```

// Раздел 7 //////////////////////////////////////

```
// Включение новой стены в передний или задний
// список корневой вершины
if (dot_wall_1 >= 0)
{
    // Поместить стену в передний список
    if (root->front==NULL)
    {
        // Это первый узел
        root->front = temp_wall;
        front_wall = root->front;
        front_wall->link = NULL;
    } // if
    else
    {
        // Это n-й узел
        front_wall->link = temp_wall;
        front_wall = front_wall->link;
        front_wall->link = NULL;
    } // else
} // if положительно
else if (dot_wall_1 < 0)
{
    // Поместить данную стену в задний список
    if (root->back==NULL)
    {
        // Это первый узел
        root->back = temp_wall;
        back_wall = root->back;
        back_wall->link = NULL;
    } // if
    else
    {
        // Это n-й узел
        back_wall->link = temp_wall;
        back_wall = back_wall->link;
        back_wall->link = NULL;
    } // else
} // if отрицательно
```

// Раздел 8 //////////////////////////////////////

```
// Обработка второй стены...

// Выделение памяти для стены
temp_wall =
    (BSPNODEV1_PTR)malloc(sizeof(BSPNODEV1));

// Задание связей
temp_wall->front = NULL;
temp_wall->back = NULL;
```

```

temp_wall->link - NULL;

// Нормаль к многоугольнику прежняя
temp_wall->wall.normal= next_wall->wall.normal;
temp_wall->wall.nlength= next_wall->wall.nlength;

// Цвет многоугольника прежний
temp_wall->wall.color = next_wall->wall.color;

// Материал многоугольника прежний
temp_wall->wall.mati = next_wall->wall.mati;

// Текстура многоугольника прежняя
temp_wall->wall.texture= next_wall->wall.texture;

// Атрибуты многоугольника прежние
temp_wall->wall.attr - next_wall->wall.attr;

// Состояния многоугольника прежние
temp_wall->wall.state - next_wall->wall.state;

// Добавление члена, указывающего на расщепление
temp_wall->id - next_wall->id + WALL_SPLIT_ID;

// Вычисление вершин стены
for (index=0; index < 4; index++)
{
    temp_wall->wall.vlist[index].x =
        next_wall->wall.vlist[index].x;
    temp_wall->wall.vlist[index].y =
        next_wall->wall.vlist[index].y;
    temp_wall->wall.vlist[index].z -
        next_wall->wall.vlist[index].z;
    temp_wall->wall.vlist[index].w = 1;

    // Копирование атрибутов вершин, координат
    // текстуры, нормали
    temp_wall->wall.vlist[index].attr=
        next_wall->wall.vlist[index].attr;
    temp_wall->wall.vlist[index].n -
        next_wall->wall.vlist[index].n;
    temp_wall->wall.vlist[index].t =
        next_wall->wall.vlist[index].t;
} // forindex

// Модифицирование вершин 0 и 3 для отражения
// точки пересечения. Координата y при данном
// разбиении стены остается неизменной
temp_wall->wall.vlist[0].x = xi;
temp_wall->wall.vlist[0].z = zi;

temp_wall->wall.vlist[3].x - xi;
temp_wall->wall.vlist[3].z - zi;

```

```

// Включение новой стены в передний или задний
// список корневой вершины
if (dot_wall_2 >= 0)
{
    // Помещение стены в передний список
    if (root->front==NULL)
    {
        // Это первый узел
        root->front = temp_wall;
        front_wall = root->front;
        front_wall->link = NULL;
    } // if
    else
    {
        // Это n-й узел
        front_wall->link = temp_wall;
        front_wall = front_wall->link;
        front_wall->link = NULL;
    } // else
} // if положительно
else if (dot_wall_2 < 0)
{
    // Помещение стены в задний список
    if (root->back==NULL)
    {
        // Это первый узел
        root->back = temp_wall;
        back_wall = root->back;
        back_wall->link = NULL;
    } // if
    else
    {
        // Это n-й узел
        back_wall->link = temp_wall;
        back_wall = back_wall->link;
        back_wall->link = NULL;
    } // else
} // if отрицательно

// Раздел 9 //////////////////////////////////////

// Мы выполнили расщепление стены, поэтому ее
// можно удалить
temp_wall = next_wall;
next_wall = next_wall->link;

// Освобождение памяти
free(temp_wall);
} // else
} // while

// Раздел 10 //////////////////////////////////////

```

```
// Рекурсивная обработка передних и задних
// стен/поддеревьев
Bsp_Build_Tree(root->front);
Bsp_Build_Tree(root->back);

} // Bsp_Build_Tree
```

Основные этапы работы данной функции выделены в разделы. Перечислим, что делается в каждом разделе.

- Раздел 1. Идет проверка, не имеет ли **текущий** указатель BSPNODEVI значение NULL. Если нет, извлекаются вершины плоскости разделения.
- Раздел 2. Две конечные точки первой стены, находящейся в связанном списке ниже стены, задающей плоскость разделения, используются для определения ее положения относительно плоскости разделения (при помощи скалярных произведений).
- Раздел 3. Обрабатывается частный случай наличия **общих** конечных точек и устанавливаются флаги для передачи соответствующей информации в следующую фазу программы.
- Раздел 4. На основании знаков скалярных произведений определено, что тестируемая стена находится перед плоскостью разделения. Данная стена добавляется к переднему связанному списку.
- Раздел 5. На основании знаков скалярных произведений определено, что тестируемая стена находится позади плоскости разделения. Данная стена добавляется к заднему связанному списку.
- Раздел 6. Тестируемая стена делится плоскостью разделения. Вычисляются координаты двух половин.
- Раздел 7. Если выполнен раздел 6, первая половина стены включается в передний или задний связанный список.
- Раздел 8. Если выполнен раздел 6, вторая половина стены включается в передний или задний связанный **список**.
- Раздел 9. Осуществляется доступ к следующей подлежащей обработке стене и процесс продолжается.
- Раздел 10. После **того** как будут обработаны все стены списка, данной функцией рекурсивно обрабатываются полученные в результате работы передний и задний списки.

Когда функция завершает свою работу, **BSP-дерево** построено, при этом связи исходного списка нарушены. Потеря памяти не происходит, однако использовать элемент link для обхода списка стен нельзя. Теперь, когда **BSP-дерево** построено, рассмотрим, как происходит его отображение.

## Обход и отображение BSP-дерева

Мы уже обсуждали алгоритм обхода BSP-дерева. Но как встроить его в наш графический конвейер? Ответ заключается в том, что по мере обхода мы будем включать многоугольники в порядке их посещения в список визуализации, а затем просто **передавать** данный список оставшимся модулям конвейера, как любой другой объект. Естественно, что при обходе BSP-дерева и включении каждого многоугольника в список визуализации

нам необходимо разбивать четырехугольник на два треугольника и вычислять соответствующие вершины, координаты текстуры и т.д. для двух треугольников, **составляющих** исходный четырехугольник, как показано на рис. 13.24.

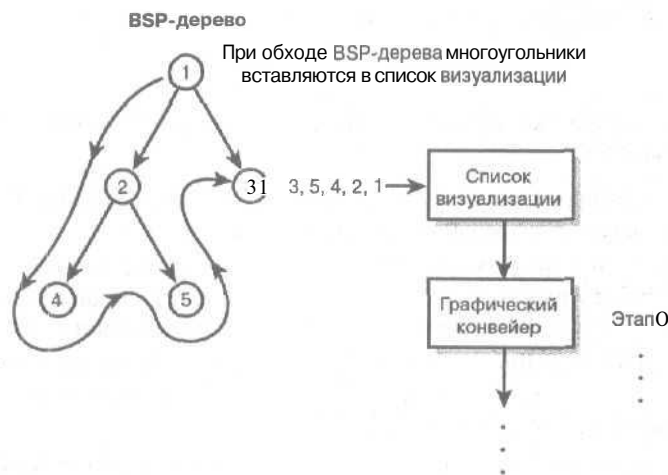


Рис. 13.24. Обход *BSP*-дерева и включение в глобальный список визуализации

Неоспоримым преимуществом является то, что в ходе визуализации нам не нужно производить сортировку или *Z*-буферизацию. Нужно только убедиться, что составляющие *BSP*-дерево многоугольники включаются в список с помощью модифицированного симметричного обхода, что обеспечивает порядок их включения от дальних к ближним (как в алгоритме художника). Порядок следования многоугольников будет правильным, поскольку они включаются в список в результате модифицированного рекурсивного последовательного обхода *BSP*-дерева, что гарантирует соблюдение надлежащего порядка для любой точки наблюдения. Ниже приведена функция, осуществляющая обход *BSP*-дерева и добавляющая многоугольники в список визуализации.

```
void Bsp_Insertion_Traversal_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, // Список визуализации
    BSPNODEV1_PTR root,          // Корень bsp-дерева
    CAM4DV1_PTR cam,             // Текущая камера
    int insert_local=0)           // Используемые вершины
{
    // Функция преобразует bsp-дерево целиком в список
    // поверхностей и затем вставляет видимые, активные, не
    // отсеченные, не отброшенные многоугольники в список
    // визуализации. Обратите внимание на флаг insert_local,
    // с помощью которого можно задавать, какой список
    // вершин используется, vlist_local или vlist_trans.
    // Если установить insert_local равным 1, то это
    // позволяет включать bsp-дерево напрямую, не
    // преобразовывая. Значение по умолчанию равно 0, это
    // означает, что объект будет включаться только после
    // преобразования локальных координат в мировые
```

```

// Функция совершает рекурсивный обход дерева в порядке
// от дальних многоугольников к ближним по отношению к
// точке наблюдения, в которой установлена камера. Для
// корректной работы bsp-дерево должно быть в мировых
// координатах

// В ходе выполнения функция тестирует точку наблюдения
// относительно текущей стены bsp, затем, в зависимости
// от того, с какой стороны находится точка наблюдения,
// выполняется определенный алгоритм. Далее производится
// обычный поиск, как в любом последовательном методе, с
// дополнительными блоками для обработки и добавления
// каждого узла в список многоугольников в
// соответствующее время

static VECTOR4D test_vector;
static float dot_wall;

// Раздел 1 //////////////////////////////////////

Write_Error("\nEntering "
            "Bsp_Insertion_Traversal_RENDERLIST4DV2()...");
Write_Error("\nTesting root...");

// Это тупик?
if (root==NULL)
{
    Write_Error("\nRoot was null...");
    return;
} // if

Write_Error("\nRoot was valid...");

// Проверка, с какой стороны от текущей стены находится
// точка наблюдения
VECTOR4D_Build(&root->wall.vlist[0].v,
               &cam->pos, &test_vector);

// Вычисляется скалярное произведение тестируемого
// вектора и нормали к поверхности и анализируется его
// знак
dot_wall=VECTOR4D_Dot(&test_vector,&root->wall.normal);

Write_Error("\nTesting dot product...");

// Раздел 2 //////////////////////////////////////

// Если знак скалярного произведения положителен, то
// наблюдатель находится с лицевой стороны от текущей
// стены, поэтому сначала рекурсивно обрабатываются
// стены, находящиеся позади, а затем - впереди данной
// стены; в противном случае порядок действий обратный
if (dot_wall > 0)

```

```

{
// Наблюдатель перед стеной
Write_Error("\nDot > 0, front side...");

// Обработка заднего поддерева для данной стены
Bsp_Insertion_Traversal_RENDERLIST4DV2(
    rend_list, root->back, cam, insert_local);

// Разбиение четырехугольника на 2 треугольника для
// включения в список
POLYF4DV2 poly1, poly2;

// Единственное отличие между POLYF4DV2 и POLYF4DV2Q
// в том, что последний имеет 4 вершины вместо 3,
// поэтому нужно создать 2 треугольника из одного
// четырехугольника :)

// Копируем важные поля
poly1.state = root->wall.state;
// Информация о состоянии
poly1.attr = root->wall.attr;
// Физические атрибуты многоугольника
poly1.color = root->wall.color;
// Цвет многоугольника
poly1.texture = root->wall.texture;
// Указатель на информацию о текстуре для простого
// отображения текстуры
poly1.mati = root->wall.mati;
// Индекс материала; (-1) для отсутствия материала
// (новый параметр)
poly1.nlength = root->wall.nlength;
// длина нормали к многоугольнику, если она не
// нормализована (новый параметр)
poly1.normal = root->wall.normal;
// Обобщенная нормаль к многоугольнику (новый
// параметр)

poly2.state = root->wall.state;
// Информация о состоянии
poly2.attr = root->wall.attr;
// Физические атрибуты многоугольника
poly2.color = root->wall.color;
// Цвет многоугольника
poly2.texture = root->wall.texture;
// Указатель на информацию о текстуре для простого
// отображения текстуры
poly2.mati = root->wall.mati;
// Индекс материала; (-1) для отсутствия материала
// (новый параметр)
poly2.nlength = root->wall.nlength;
// Длина нормали к многоугольнику, если она не
// нормализована (новый параметр)
poly2.normal = root->wall.normal;
// Обобщенная нормаль к многоугольнику(новый

```

```

// параметр)

// Теперь перейдем к вершинам, которые в настоящий
// момент выглядят следующим образом:
//vO v1
//
//
//v3 v2
// Мы же хотим создать два треугольника, имеющих
// следующий вид:
// poly 1      poLy2
//
//vO v1      v1
//
//
//
// v3      v3 v2
//
// В соответствии с соглашениями о том, что вершины
// перечисляются по часовой стрелке, порядок вершин
// в многоугольнике 1 - vO,v,v3, а в многоугольнике
// 2 - v1,v2,v3
if (insert_local==1)
{
    // Многоугольник 1
    poly1.vlist[0] - root->wall.vlist[0];
    poly1.tvlist[0] = root->wall.vlist[0];

    poly1.vlist[1] - root->wall.vlist[1];
    poly1.tvlist[1] - root->wall.vlist[1];

    poly1.vlist[2] - root->wall.vlist[3];
    poly1.tvlist[2] = root->wall.vlist[3];

    // Многоугольник 2
    poly2.vlist[0] - root->wall.vlist[1];
    poly2.tvlist[0] - root->wall.vlist[1];

    poly2.vlist[1] - root->wall.vlist[2];
    poly2.tvlist[1] - root->wall.vlist[2];

    poly2.vlist[2] = root->wall.vlist[3];
    poly2.tvlist[2] - root->wall.vlist[3];
} // if
else
{
    // Многоугольник 1
    poly1.vlist[0] = root->wall.vlist[0];
    poly1.tvlist[0] - root->wall.tvlist[0];

    poly1.vlist[1] - root->wall.vlist[1];
    poly1.tvlist[1] - root->wall.tvlist[1];

    poly1.vlist[2] - root->wall.vlist[3];

```

```

poly1.tvlist[2]=root->wall.tvlist[3];

// Многоугольник 2
poly2.vlist[0] = root->wall.vlist[1];
poly2.tvlist[0] = root->wall.tvlist[1];

poly2.vlist[1] = root->wall.vlist[2];
poly2.tvlist[1] = root->wall.tvlist[2];

poly2.vlist[2] = root->wall.vlist[3];
poly2.tvlist[2] = root->wall.tvlist[3];
} // if

Write_Error("\nInserting polygons...");

// Включение многоугольников в список визуализации
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly1);
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly2);

// Обработка переднего поддерева заданной стены
Bsp_Insertion_Traversal_RENDERLIST4DV2(
    rend_list, root->front.cam, insert_local);

} // else

// Раздел 3 //////////////////////////////////////

else
{
    // Камера находится позади данной стены
    Write_Error("\nDot < 0, back side...");

    // Обработка переднего поддерева данной стены
    Bsp_Insertion_Traversal_RENDERLIST4DV2(
        rend_list, root->front.cam, insert_local);

    // Разбиение четырехугольника на 2 треугольника для
    // включения в список
    POLYF4DV2 poly1, poly2;

    // Единственное отличие между POLYF4DV2 и POLYF4DV2Q
    // состоит в том, что последний имеет 4 вершины
    // вместо 3, поэтому мы собираемся создать 2
    // треугольника из одного четырехугольника :)

    // Копирование важных полей
    poly1.state = root->wall.state;
    // Информация о состоянии
    poly1.attr = root->wall.attr;
    // Физические атрибуты многоугольника
    poly1.color = root->wall.color;
    // Цвет многоугольника
    poly1.texture = root->wall.texture;
    // Указатель на информацию о текстуре для простого

```

```

// отображения текстуры
poly1.mati = root->wall.mati;
// Индекс материала; (-1) для отсутствия материала
// (новый параметр)
poly1.nlength = root->wall.nlength;
// Длина нормали к многоугольнику, если она не
// нормализована (новый параметр)
poly1.normal = root->wall.normal;
// Обобщенная нормаль к многоугольнику (новый
// параметр)

poly2.state = root->wall.state;
// Информация о состоянии
poly2.attr = root->wall.attr;
// Физические атрибуты многоугольника
poly2.color = root->wall.color;
// Цвет многоугольника
poly2.texture = root->wall.texture;
// Указатель на информацию о текстуре для простого
// отображения текстуры
poly2.mati = root->wall.mati;
// Индекс материала; (-1) при отсутствии материала
// (новый параметр)
poly2.nlength = root->wall.nlength;
// Длина нормали к многоугольнику, если не
// нормализована (новый параметр)
poly2.normal = root->wall.normal;
// Обобщенная нормаль к многоугольнику (новый
// параметр)

// Теперь перейдем к вершинам, которые в настоящий
// момент выглядят следующим образом:
//v0 v1
//
//
//v3 v2
//
// Мы же хотим создать 2 треугольника, которые
// выглядят так:
// poly 1    poly2
//v0 v1    v1
//
//
//
//v3    v3 v2
//
// Порядок перечисления вершин (по часовой стрелке,
// в соответствии с правилом) - для poly 1:
// v0,v1,v3; для poly2: v1,v2,v3
if (insert_local==1)
{
    // Многоугольник 1
    poly1.vlist[0] = root->wall.vlist[0];
    poly1.tvlist[0] = root->wall.vlist[0];
}

```

```

poly1.vlist[1] - root->wall.vlist[1];
poly1.tvlist[1] = root->wall.vlist[1];

poly1.vlist[2] - root->wall.vlist[3];
poly1.tvlist[2] = root->wall.vlist[3];

// Многоугольник 2
poly2.vlist[0] - root->wall.vlist[1];
poly2.tvlist[0] - root->wall.vlist[1];

poly2.vlist[1] = root->wall.vlist[2];
poly2.tvlist[1] - root->wall.vlist[2];

poly2.vlist[2] = root->wall.vlist[3];
poly2.tvlist[2] * root->wall.vlist[3];
} // if
else
{
    // Многоугольник 1
    poly1.vlist[0] = root->wall.vlist[0];
    poly1.tvlist[0] = root->wall.tvlist[0];

    poly1.vlist[1] - root->wall.vlist[1];
    poly1.tvlist[1] - root->wall.tvlist[1];

    poly1.vlist[2] - root->wall.vlist[3];
    poly1.tvlist[2] - root->wall.tvlist[3];

    // Многоугольник 2
    poly2.vlist[0] - root->wall.vlist[1];
    poly2.tvlist[0] - root->wall.tvlist[1];

    poly2.vlist[1] - root->wall.vlist[2];
    poly2.tvlist[1] = root->wall.tvlist[2];

    poly2.vlist[2] - root->wall.vlist[3];
    poly2.tvlist[2] = root->wall.tvlist[3];
} // else

Write_Error("\nInserting polygons...");

// Включение многоугольников в список визуализации
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly1);
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly2);

// Обработка заднего поддерева заданной стены
Bsp_Insertion_Traversal_RENDERLIST4DV2(
    rend_list, root->back, cam, insert_local);
} // else

Write_Error("\nExiting "
    "Bsp_Insertion_Traversal_RENDERLIST4DV2{ }...");
} // Bsp_Insertion_Traversal_RENDERLIST4DV2

```

Как обычно, реализация функции значительно отличается от теоретического описания. Как нетрудно убедиться, приведенная функция гораздо сложнее, чем предложенная ранее версия с использованием псевдокода. Рассмотрим ее разделы.

- Раздел 1. Идет проверка, не является ли текущий узел пустым (NULL). Если это так, производится выход; в противном случае определяется положение точки наблюдения относительно текущего узла при помощи скалярного произведения.
- Раздел 2. Знак полученного в разделе 1 скалярного произведения используется для определения, с какой стороны от плоскости находится наблюдатель. Исходя из этого, выбирается один из двух методов обхода, идентичных во всем, за исключением рекурсивных вызовов. Если точка наблюдения находится перед плоскостью узла, производится рекурсивный вызов для обработки заднего списка стен, расщепление многоугольника на два треугольника и включение их в список визуализации с последующим рекурсивным вызовом функции для обработки переднего списка.
- Раздел 3. Если точка наблюдения находится позади плоскости узла, сначала производится рекурсивный вызов для обработки переднего списка стен. Затем рассматриваемый многоугольник расщепляется на два треугольника, которые добавляются к списку визуализации. После этого производится рекурсивный вызов функции для обработки заднего списка.

После вызова функции `Bsp_Insertion_Traversal_RENDERLIST4DV2` BSP полностью вносится в список визуализации, и мы можем просто продолжать использовать конвейер визуализации, как это делается после включения любого объекта.

При этом необходимо рассмотреть ряд вопросов. Во-первых, необходимо отметить, что функция имеет флаг `insert_local`, который определяет, какой из двух списков вершин, локальный (`vlist[]`) или преобразованный (`tvlist[]`) используется при включении стены в список многоугольников. BSP-дерево находится в локальном пространстве, но поскольку это сетка некоторого уровня, локальное пространство может совпадать с мировым, и оба массива могут иметь одни и те же данные. Однако если это не так, и вы хотите выполнить преобразование BSP-дерева, нужно просто совершить его обход (в произвольном порядке) и выполнить необходимые преобразования. Например, если вы хотите выполнить перенос всех вершин BSP, можно написать что-то наподобие следующего.

```
void Bsp_Translate(BSPNODEV1 PTR root, VECTOR4D_PTR trans)
```

```
{
```

```
    // Данная функция выполняет перенос всех стен,  
    // образующих данное bsp-дерево
```

```
    // Это рекурсивная функция. В действительности мы в ней  
    // не нуждаемся, но это хороший пример того, как можно  
    // преобразовывать BSP-дерево и древовидные структуры  
    // с помощью рекурсии. Обратите внимание на  
    // преобразование из локальных координат в мировые
```

```
    static int index; // переменная цикла
```

```
    // Проверка, не достигнут ли тупик  
    if (root==NULL)  
        return;
```

```
    // Преобразование самого дальнего поддерев
```

```

Bsp_Translate(root->back, trans);

// Итеративный перебор всех вершин текущей стены и их
// преобразование
for (index=0; index < 4; index++)
{
    // Выполнение преобразования
    root->wall.tvlist[index].x -
        root->wall.vlist[index].x + trans->x;
    root->wall.tvlist[index].y =
        root->wall.vlist[index].y + trans->y;
    root->wall.tvlist[index].z =
        root->wall.vlist[index].z + trans->z;
} // for index

// Преобразование самого ближнего поддерева
Bsp_Translate(root->front, trans);

} // Bsp_Translate

```

Нужно просто **передать** функции корень BSP и вектор преобразования, и все дерево будет преобразовано.

Более общая функция преобразования совершает обход **BSP** и применяет к его вершинам матричное преобразование. Кроме того, как и в других случаях, можно указать, к каким координатам применяется преобразование: локальным, преобразованным или локальным, скопированным в преобразованные. Указанные возможности обеспечивает следующая функция.

```

void Bsp_Transform(
    BSPNOOE1_PTR root, // Корень bsp-дерева
    MATRIX4X4_PTR mt, // Матрица преобразования
    int coord_select) // Выбор преобразуемых координат
{
    // Данная функция совершает обход bsp-дерева и применяет
    // матрицу преобразования к каждому узлу, Функция
    // является рекурсивной и использует симметричный обход.
    // Допустимы также прямой и обратный порядок обхода
    // вершин

    // Проверка, не достигнут ли конец
    if (root==NULL)
        return;

    // Преобразование заднего поддерева
    Bsp_Transform(root->back, mt, coord_select);

    // Итерации по всем вершинам текущей стены и
    // преобразование в координаты камеры

    // Какие именно координаты должны быть преобразованы?
    switch(coord_select)
    {
        case TRANSFORM_LOCAL_ONLY:
            {

```

```

// Преобразование каждой локальной вершины
// сетки объекта на месте
for (int vertex = 0; vertex < 4; vertex++)
1
    POINT4D presult;
    // Преобразование точки
    Mat_Mul_VECTOR4D_4X4(
        &root->wall.vlist[vertex].v,
        mt, &presult);
    // Запись результата
    VECTOR4D_COPY(
        &root->wall.vlist[vertex].v,
        &presult);
    // Преобразование нормали к вершине
    if (root->wall.vlist[vertex].attr &
        VERTEX4DTV1_ATTR_NORMAL)
    {
        // Преобразование нормали
        Mat_Mul_VECTOR4D_4X4(
            &root->wall.vlist[vertex].n,
            mt &presult);
        // Запись результата
        VECTOR4D_COPY(
            &root->wall.vlist[vertex].n,
            &presult);
    } // if
} // for
} break;

case TRANSFORM_TRANS_ONLY:
{
    // Преобразование каждой "преобразованной"
    // вершины сетки объекта на месте.
    // Напоминаю, что назначение массива
    // vlist_trans[] состоит в накоплении
    // преобразований
    for (int vertex = 0; vertex < 4; vertex++)
    {
        POINT4D presult;
        // Преобразование точки
        Mat_Mul_VECTOR4D_4X4(
            &root->wall.tvlist[vertex].v, mt,
            &presult);
        // Сохранение результата
        VECTOR4D_COPY(
            &root->wall.tvlist[vertex].v,
            &presult);

        // Преобразование нормали к вершине
        if (root->wall.tvlist[vertex].attr &
            VERTEX4DTV1_ATTR_NORMAL)
        {
            // Преобразование нормали
            Mat_Mul_VECTOR4D_4X4(

```

```

        &root->wall.tvlist[vertex].n,
        mt &presult);

    // Запись результата
    VECTOR4D_COPY(
        &root->wall.tvlist[vertex].n,
        &presult);
    } // if
} // for
} break;

case TRANSFORM_LOCAL_TO_TRANS:
{
    // Преобразование каждой локальной вершины
    // сетки объекта и сохранение результата в
    // "трансформированном" списке вершин
    for (int vertex=0; vertex < 4; vertex++)
    {
        POINT4D presult;
        //Преобразование точки
        Mat_Mul_VECTOR4D_4X4(
            &root->wall.vlist[vertex].v, mt
            &root->wall.tvlist[vertex].v);

        //Преобразование нормали вершины
        if (root->wall.tvlist[vertex].attr
            & VERTEX4DTV1_ATTR_NORMAL)
        {
            // Преобразование нормали
            Mat_Mul_VECTOR4D_4X4(
                &root->wall.vlist[vertex].n, mt
                &root->wall.tvlist[vertex].n);
        } // if

    } // for
} break;

default:break;

} // switch

// Преобразование переднего поддерева
Bsp_Transform(root->front, mt coord_select);

} // Bsp_Insertion_Traversal_RENDERLIST4DV2

```

При вызове функции `Bsp_Insertion_Traversal_RENDERLIST4DV2()` ей передаются корень BSP-дерева, матрица преобразования и флаг выбора преобразования. Функция рекурсивно выполняет нужное преобразование и (мы надеемся) возвращает результат. Однако в большинстве случаев вам не понадобится переносить, вращать или преобразовывать BSP, поскольку, по всей вероятности, BSP будет представлять данные статического внутреннего или внешнего уровня, где единственным движущимся предметом является камера.

Чтобы освободить память, выделенную под BSP-дерево, используется приведенная ниже функция удаления, которая совершает обход BSP и освобождает занимаемую им память.

```

void Bsp_Delete(BSPNODEV1_PTR root)
{
    // Данная функция рекурсивно удаляет все узлы bsp-дерева
    // и возвращает память операционной системе

    BSPNODEV1_PTR temp_wall; // Временная стена

    // Проверка, не достигнут ли конец
    if (root==NULL)
        return;

    // Удаление заднего поддеревя
    Bsp_Delete(root->back);

    // Удаление текущего узла; но прежде необходимо
    // сохранить переднее поддеревя
    temp_wall = root->front;

    // Освобождение памяти
    free(root);

    // Присвоение корню сохраненного переднего поддеревя
    root = temp_wall;

    // Удаление переднего поддеревя
    Bsp_Delete(root);
} // Bsp_Delete

```

Достаточно просто вызвать функцию `Bsp_Delete()`, передав ей корень BSP-дерева, и это дерево будет рекурсивно удалено.

Наконец, ниже приводится функция, которая позволит распечатать BSP при необходимости диагностики.

```

void Bsp_Print(BSPNODEV1_PTR root)
{
    // Данная функция выполняет рекурсивный
    // последовательный обход BSP-дерева и печатает
    // результаты в файл

    // Проверка, является ли данный дочерний узел пустым
    if (root==NULL)
    {
        Write_Error("\nReached NULL node returning...");
        return;
    } // if

    // Поиск левого дерева (задние стены)
    Write_Error("\nTraversing back sub-tree...");

    // Рекурсивный вызов
    Bsp_Print(root->back);

    // Посещение узла

```

```

Write_Error("\n\nWall ID # %d", root->id);

Write_Error("\nstate = %d",
    root->wall.state); // Состояние
Write_Error("\nattr = %d",
    root->wall.attr); // Атрибуты
    // многоугольника
Write_Error("\ncolor = %d",
    root->wall.color); // Цвет многоугольника
Write_Error("\ntexture = %x",
    root->wall.texture); // Указатель на
    // информацию о
    // текстуре
Write_Error("\nmati = %d",
    root->wall.mati); // Индекс материала;
    // (-1J для отсутствия
    // материала

Write_Error("\nVertex 0: (%f,%f,%f,%f)",
    root->wall.vlist[0].x,
    root->wall.vlist[0].y,
    root->wall.vlist[0].z,
    root->wall.vlist[0].w);

Write_Error("\nVertex 1: (%f,%f,%f,%f)",
    root->wall.vlist[1].x,
    root->wall.vlist[1].y,
    root->wall.vlist[1].z,
    root->wall.vlist[1].w);

Write_Error("\nVertex 2: (%f,%f,%f,%f)",
    root->wall.vlist[2].x,
    root->wall.vlist[2].y,
    root->wall.vlist[2].z,
    root->wall.vlist[2].w);

Write_Error("\nVertex 3: (%f,%f,%f,%f)",
    root->wall.vlist[3].x,
    root->wall.vlist[3].y,
    root->wall.vlist[3].z,
    root->wall.vlist[3].w);

Write_Error("\nNormal (%f,%f,%f,%f), length=%f",
    root->wall.normal.x,
    root->wall.normal.y,
    root->wall.normal.z,
    root->wall.nlength);

Write_Error("\nTextCoords (%f,%f)",
    root->wall.vlist[1].u0,
    root->wall.vlist[1].v0);

Write_Error("\nEnd wall data\n");

```

```
// Поиск правого дерева (передние стены)
Write_Error("\nTraversing front sub-tree..");

Bsp_Print(root->front);

} // Bsp_Print
```

Таким образом, для печати BSP достаточно вызвать функцию `Bsp_Print()`, передав ей корень дерева, однако необходимо убедиться, что предварительно с помощью вызова функции `Open_Error_File()` был открыт канал для вывода ошибок. На этом рассмотрение BSP-системы практически завершено. Осталось только показать, как включить ее в наш графический конвейер.

## Интеграция BSP-дерева в графический конвейер

Включить технологию BSP-дерева в графический конвейер не слишком сложно. Мы будем присоединять BSP-систему так же, как ранее уже присоединяли систему Z-буферизации. Нам не нужны никакие функции объектов, поскольку BSP-дерево само содержит всю геометрию для статических компонентов игрового пространства. BSP-объект трактуется как и любой другой объект, за одним исключением: мы должны включить BSP в список визуализации *первым* и мы *не должны* выполнять Z-сортировку списка визуализации, поскольку это нарушит заданный BSP порядок.

Если вы не заинтересованы в использовании свойства BSP-дерева обеспечивать порядок посещения вершин от дальних к ближним и хотите использовать его для выявления столкновений или отбраковки (о чем мы поговорим *позднее*), тогда можно включать BSP-дерево в список визуализации в произвольном порядке (*раньше* или *позже* других объектов) и выполнять Z-сортировку и/или Z-буферизацию. Фактически в 99% всех основанных на аппаратном обеспечении игр BSP-деревья используются только для выявления столкновений и отбраковки, а не для вычисления порядка визуализации. Однако в нашей демонстрационной версии мы собираемся использовать BSP, чтобы показать, что с *помощью* последовательного обхода BSP от дальних многоугольников к ближним можно получить точный порядок визуализации.

Чтобы включить BSP в графический конвейер, необходимо выполнить единственный вызов функции `Bsp_Insertion_Traversal_RENDERLIST4DV2()` в верхней части цикла визуализации, а том же месте, где производятся вызовы для включения объектов в список визуализации. Однако при включении BSP-дерева необходимо учитывать его упорядоченность, так что все остальные объекты, добавленные в список визуализации, будут впереди объектов дерева, поскольку никакая дополнительная сортировка не предусматривается. Можно, конечно, прибегнуть к Z-буферизации, однако это нивелирует весь смысл применения BSP в данном примере (ведь мы хотим именно с его *помощью* достичь упорядочения визуализации). Ниже приводится порядок операций для использования функций BSP в конвейере визуализации (это фрагмент демонстрационной программы, из которого удалено все лишнее, чтобы вы могли увидеть порядок главных вызовов функций).

```
// Запустит таймер
Start_Clock();

// Очистка поверхности вывода
DDDraw_Fill_Surface(lpddsback, 0);

// здесь производится считывание клавиатуры и других
// устройств
DInput_Read_Keyboard();
```

```

// Здесь размещается логика игры...

// Сброс списка визуализации
Reset_RENDERLIST4DV2(&rend_list);

// Генерация матрицы преобразования к координатам камеры
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// включение bsp в список визуализации
Bsp_Insertion_Traversal_RENDERLIST4DV2 (&rend_list,
                                         bsp_root, &cam, 1);

// Преобразование мировых координат в координаты камеры
World_To_Camera_RENDERLIST4DV2(&rend_list, &cam);

// Отсечение многоугольников
Clip_Polys_RENDERLIST4DV2(
    &rend_list, &cam,
    ((x_clip_mode == 1) ? CLIP_POLY_X_PLANE: 0) |
    ((y_clip_mode == 1) ? CLIP_POLY_Y_PLANE: 0) |
    ((z_clip_mode == 1) ? CLIP_POLY_Z_PLANE: 0) );

// Освещение сцены
if (lighting mode==1)
{
    Transform_LIGHTSV2(lights2, 4, &cam.mcam,
                      TRANSFORM_LOCAL_TO_TRANS);
    Light_RENDERLIST4DV2_World2_16(&rend_list,
                                   &cam, lights2, 4);
} // if

// Аксонометрическое преобразование координат
Camera_To_Perspective_RENDERLIST4DV2(&rend_list, &cam);

// Преобразование в экранные координаты
Perspective_To_Screen_RENDERLIST4DV2(&rend_list, &cam);

// Блокирование вторичной поверхности
DDraw_Lock_Back_Surface();

// Настройка контекста визуализации при отсутствии
// Z-буферизации
rc.attr = RENDER_ATTR_NOBUFFER |
          RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;

rc.video_buffer - back_buffer;
rc.lpitch       = back_lpitch;
rc.mip_dist     = 0;
rc.zbuffer      = (UCHAR *)zbuffer.zbuffer;
rc.zpitch       - WINDOW_WIDTH*4;
rc.rend_list    - &rend_list;
rc.texture_dist - 0;
rc.alpha_override - -1;

// Визуализация сцены
Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(&rc);

```

```
// Разблокирование вторичной поверхности  
DDraw_Unlock_Back_Surface();
```

```
// Переключение поверхностей  
DDraw_Flip2();
```

Как видно из данного фрагмента, BSP включается в конвейер с помощью единственного вызова; все остальные вызовы осуществляются как обычно. Окончательная версия демонстрационной программы содержит как потолочную, так и напольную сетку. Они фактически являются объектами **OBJEKT4DV2** и включаются в список визуализации самыми первыми, за ними следует BSP, поскольку, по определению, пол и потолок всегда находятся под стенами; или, иными словами, стены всегда загораживают полы и потолок. Кроме того, полы и потолки не нужно сортировать, поскольку все они компланарны (мы вернемся к этому позднее).

## Редактор уровня

Демонстрационная программа BSP примерно на 90% состоит из кода интерфейса и содержит всего несколько обращений к BSP-функциям. Это яркий пример того, насколько сложно писать интерфейсы. Зачастую код интерфейса на порядок больше, чем сама программа. Демонстрационная программа называется **DEMOII13\_1.CPP|EXE** и позволяет нам использовать мышь для рисования вида сверху игрового уровня, состоящего из двумерных стен. На рис. 13.25 показана копия экрана программы. На нем можно выделить три области. Область справа представляет собой область управления, а слева — редактируемое пространство. Полоса меню используется для выбора различных режимов, компиляции, сохранения, загрузки и т.д.

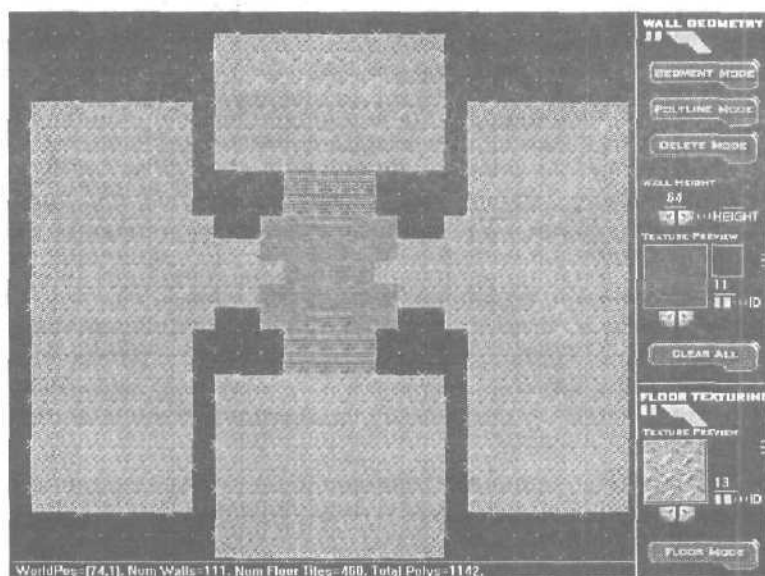


Рис. 13.25. Копия экрана при работе демонстрационной версии BSP

НА ЗАМЕТКУ

Чтобы самостоятельно скомпилировать демонстрационную программу, вам понадобятся файл **DEMOII13\_1.CPP**, а также библиотечные модули **T3LIB1-11.CPP|H** и библиотечные файлы **DirectX**. Кроме того, понадобится резервный файл для меню и т.п., который называется **DEMOII13\_1.RC**.

Прежде чем перейти к описанию использования редактора, необходимо поговорить о некоторых его ограничениях.

- Должно быть не более 256 линий.
- В процессе рисования стен я советую не делать их длину более 2–4 шагов решетки — в противном случае вы получите неприятные артефакты отображения текстуры из-за аффинного **текстурирования**.
- Старайтесь, чтобы сегменты стен не перекрывались.
- Старайтесь, чтобы уровни были простыми — данная система недостаточно надежна и сложная задача может привести к ее отказу.
- Обе программы, редактор и программа просмотра работают в оконном режиме, минимальное разрешение дисплея должно составлять 800х600х16. Глубина цвета должна быть равна 16, как и во всех других оконных приложениях в данной книге.

## Демонстрационная версия Doom

Эта демонстрационная программа мало похожа на известную игру *Doom*, но это привлекательное название, поэтому я решил им воспользоваться. Прежде чем перейти к рассмотрению кода и функций редактора, следует кратко рассмотреть, как загрузить уровень и запустить демонстрационную программу. Я предлагаю действовать **следующим** образом.

1. Запустить программу **DEMO113\_1.EXE** из ее рабочего каталога. Чтобы гарантировать нормальную работу системы, убедитесь, что ваш дисплей находится в режиме с 16-битовой цифровой палитрой и разрешением 800х600х16 или выше (я рекомендую 1024х768). Экран будет выглядеть примерно так, как показано на рис. 13.26.
2. В главном меню выберите команду **File⇒Load**. LEV File, как показано на рис. 13.27. После этого наберите в поле ввода **DOOM01.LEV** и **щелкните** на кнопке OK (рис. 13.28).
3. Редактор загрузит уровень, и на экране появится изображение, похожее на рис. 13.25. Чтобы преобразовать уровень в **BSP-дерево** и отобразить его, достаточно просто выбрать команду меню **Build⇒Compile BSP and View**. Программа скомпилирует уровень в BSP-дерево и перейдет в трехмерный режим тестирования (рис. 13.29).
4. Теперь вы находитесь в трехмерном мире. В нем можно двигаться, переключать **освещение** с помощью клавиш <A>, <I> и <P>. Если ваш компьютер достаточно быстрый (не ниже 1.5 GHz), картинка на дисплее будет достаточно живой. Попробуйте пройти сквозь стены и прогуляться вокруг игрового мира. Испытайте каркасный режим с помощью клавиши <W> и обратите внимание на счетчик **многоугольников** в каждом режиме.
5. Вернитесь в редактор с помощью клавиши <Esc>. Можно входить в трехмерный режим и возвращаться из него вновь и вновь, повторяя шаги 3–5. Это и есть основная **последовательность действий** при моделировании.

### НА ЗАМЕТКУ

Когда еще продолжалась работа над этой главой, в демонстрационной программе BSP все еще оставались неустраненные дефекты, поэтому иногда система зависала или **неправильно** отображала некоторые стены. Так что во избежание неприятностей не создавайте в ней сложную геометрию.

Теперь, получив **общее** представление о работе редактора уровня, рассмотрим используемый формат файла, элементы управления, а также использование редактора.

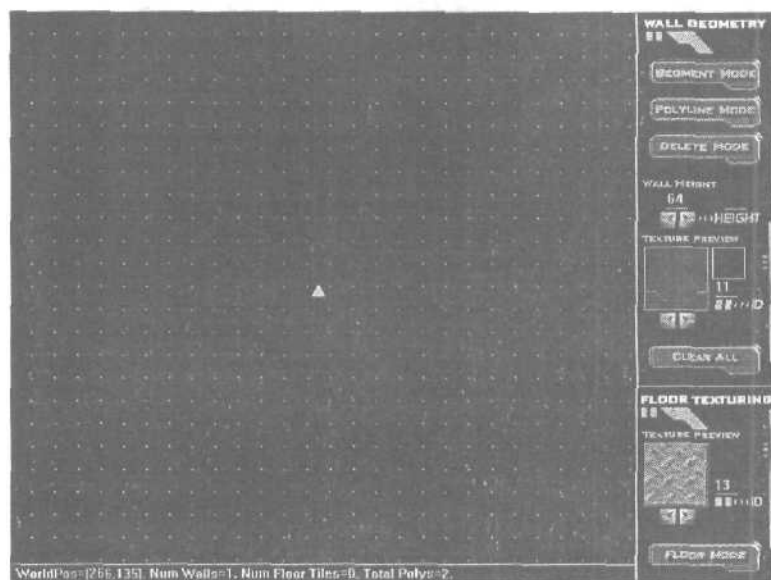


Рис. 13.26. Запуск редактора уровня

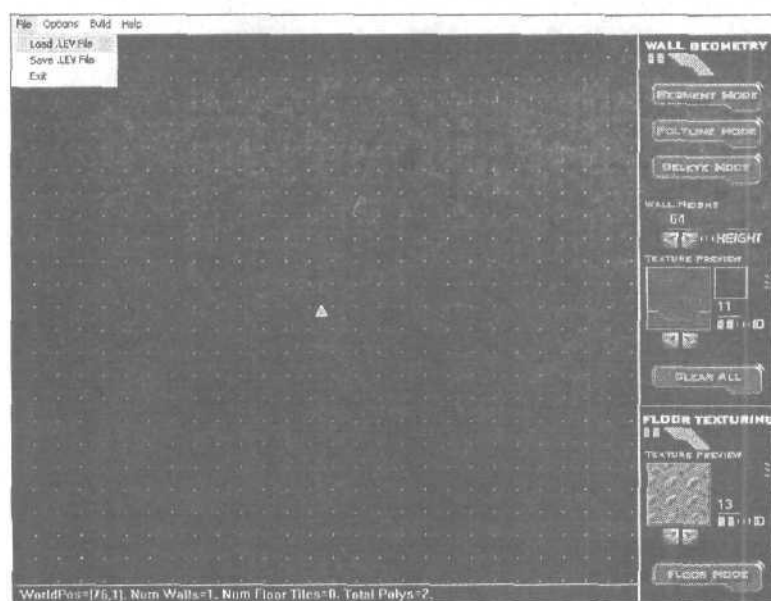


Рис. 13.27. Загрузка уровня в редактор

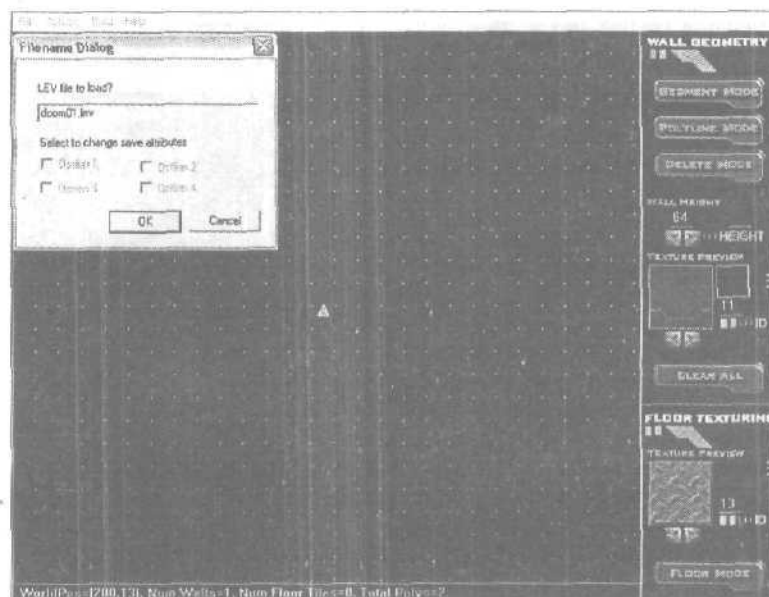


Рис. 13.28. Ввод имени файла

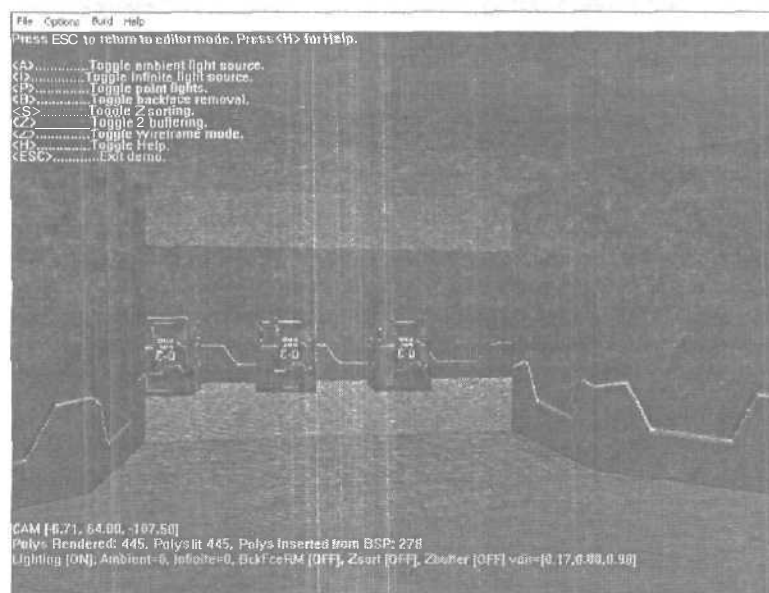


Рис. 13.29. Редактор уровня в трехмерном режиме

## Графический интерфейс редактора уровня

Когда я впервые задумал создать элементарный редактор уровня, первым побуждением было взять за основу формат *Quake*, но он слишком сложен. В нем слишком много лишнего, а я предпочитаю простые форматы, чтобы можно было понять лежащие в основе принципы, а не заниматься ненужными сложностями.

Приняв решение сделать редактор уровня, нужно было определиться, как его писать: использовать Visual Basic, Visual C++, Borland Builder, DirectX или GDI? В реальной жизни я пишу все свои программы с помощью Builder или Visual Basic, но в данном случае это потребует слишком много работы, и вы не сможете играть и дополнять программу, поэтому я решил использовать Visual C++,

Затем встал вопрос о типе приложения — оконном или полноэкранном. Для трехмерного режима необходим DirectX, но для графического интерфейса нужны кнопки и элементы управления. Я решил использовать элементы управления Windows только в меню, а все остальное делать вручную, т.е. создать собственные клавиши для элементов управления редактируемым пространством. Это оказалось проще, чем заставить Windows и DirectX совместно работать как нужно. В любом случае, полученный редактор вполне отвечает нашим запросам. Мы еще вернемся к его коду позже, а сейчас давайте перейдем к его использованию и рассмотрим, как работает каждый элемент управления.

Назначение редактора состоит в том, чтобы предоставить возможность рисовать сегменты стен в плоскости xz, которые представляют трехмерные стены в трехмерном пространстве. Он также размещает текстуры плит пола, которые в трехмерном мире представляются многоугольниками. Таким образом, с точки зрения пользователя, мы просто рисуем сегменты стен и плиты пола с различными текстурами, а затем строим BSP и рассматриваем его представление.

Изучим каждый элемент панели управления, находящейся в правой части экрана. Как видно на рис. 13.30, управляющая панель состоит из двух групп управляющих элементов:

- геометрия стен (Wall Geometry);
- текстура пола (Floor Texturing).

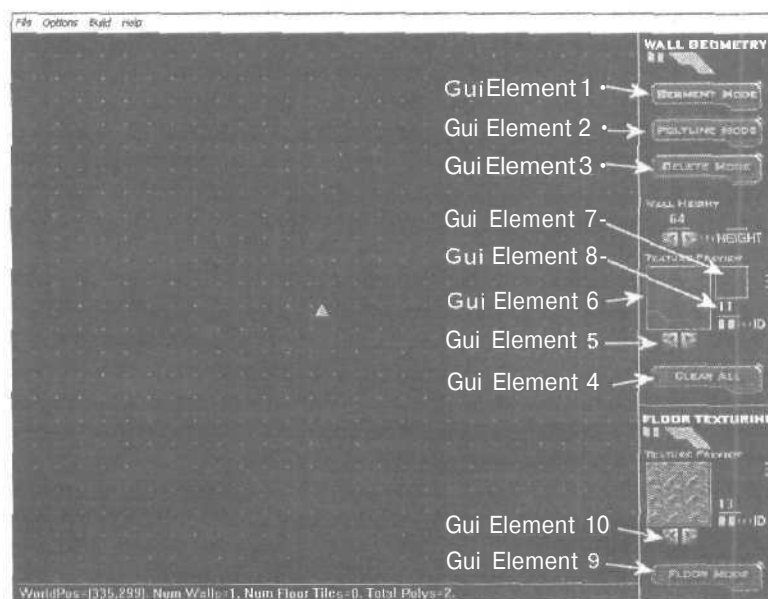


Рис. 13.30. Подробное представление интерфейса редактора уровня

Segment Mode. Рассмотрим сначала раздел геометрии стен. Элемент 1, Segment Mode, сигнализирует, что вы хотите рисовать отдельные отрезки в находящейся слева области редактирования. Щелкнув левой кнопкой мыши на данной клавише, вы переводите

те систему в режим рисования отдельных отрезков. Этот режим работает следующим образом: чтобы создать узел, достаточно установить указатель мыши в область редактирования и щелкнуть левой кнопкой. Если отпустить кнопку мыши и перемещать указатель, вы увидите, что за указателем мыши тянется линия. Нужно снова нажать на левую кнопку; в результате между точками генерируется отрезок стены. Такой режим хорош в том случае, если вы хотите рисовать стены по одной. Если вы решили закончить текущий отрезок, не закончив линию, нужно нажать на правую кнопку, и система отбросит оставшуюся часть линии.

**Polyline Mode.** Это более гибкий режим, который позволяет последовательно рисовать несколько линейных сегментов. Если выбрать элемент 2 (см. рис. 13.30) и щелкнуть на нем левой кнопкой мыши, редактор перейдет в полилинейный режим. Он работает следующим образом. Если перейти в редактируемую область и щелкнуть левой кнопкой мыши, создается стартовая точка. Затем к текущей позиции мыши потянется линия, которая будет показывать, где будет помещена следующая линия. Чтобы разместить линию, нужно просто еще раз щелкнуть левой кнопкой. В этом режиме система позволяет добавлять отрезки, не завершая текущий, как это было в режиме рисования отдельных отрезков. Чтобы прекратить процесс добавления отрезков, нужно просто щелкнуть правой кнопкой мыши.

**Delete Mode.** Следующим режимом является режим удаления стен. Чтобы перейти в этот режим, нужно подвести курсор к клавише **Delete Mode**, показанной на рис. 13.30 как элемент 3. Находясь в данном режиме, система находит ближайший к курсору отрезок в поле редактирования и подсвечивает его. Достаточно щелкнуть левой кнопкой мыши над ним, и отрезок будет удален из системы.

**Clear All.** Очередной элемент используется для того, чтобы полностью очистить систему, включая стены и полы, и вернуть все в исходное состояние. Достаточно щелкнуть на этой клавише левой кнопкой мыши, и уровень исчезнет.

**Wall Height and Texture Preview.** Мы уже знаем, как с помощью редактора рисовать линии, но что они собой представляют? Это трехмерные вертикальные стены в плоскости xz, размеры которых определяются сеткой, шаг которой равен примерно 48 единицам. Однако мы еще ничего не знаем о высоте стен или налагаемой на них текстуре. Эти значения задаются с помощью элементов 5 и 6 графического интерфейса. Элемент 5 задает высоту стены; изменить ее можно, щелкая на голубых стрелках, находящихся ниже поля значений. Я советую для начала сделать стены высотой 128 шагов координатной сетки и длиной 2 шага.

На каждую стену без повторов отображается текстура. Все текстуры имеют размеры 128x128, и в программе их имеется около 20. Текущая активная текстура высвечивается в окне предварительного просмотра текстуры, обозначенном в графическом интерфейсе как элемент 6. Чтобы изменить текстуру, нужно щелкнуть на стрелках влево и вправо под данным окном. Заметим, что все стены, нарисованные с текущим значением высоты и текстуры, изменить нельзя. Единственный способ модифицировать стену — удалить ее и нарисовать заново.

**Wall Height and Texture Scanners.** В дополнение к высоте стен и предварительного просмотра текстуры, есть еще элементы интерфейса, обозначенные номерами 7 и 8. Они показывают текстуру и высоту стены для любого отрезка в редактируемой области слева, над которым находится курсор. Например, если вы нарисовали уровень и забыли, какую текстуру наложили на некую стену или какова ее высота, достаточно указать мышью на отрезок стены, чтобы он подсветился, а сканнеры текстуры и высоты покажут соответствующие данные. Помните, эти данные нельзя изменить — для этого необходимо удалить отрезок и нарисовать его заново.

Floor Mode. Еще одной важной частью интерфейса редактора уровня является элемент 9. При вычерчивании уровня не задается геометрия пола или потолка. Чтобы добавить плитку пола, нужно выбрать режим рисования пола, щелкнув на соответствующей клавише. При этом на любую плитку в области редактирования, на которой вы щелкнете левой кнопкой, будет наложена текущая текстура, отображенная в окне текстуры пола. Таким способом можно "закрасить" пол. Если произошла ошибка, плитки пола можно стереть, щелкнув на них правой кнопкой, но для этого необходимо находиться в режиме рисования пола. Как и в случае текстуры стен, можно выбирать любые плитки пола из примерно 20 возможных, щелкая на стрелках влево и вправо под окном предварительного просмотра текстуры.

The Main Menu. Главное меню состоит из нескольких меню следующего уровня.

- File Содержит три команды: Load.LEV, Save.LEV и Exit. Load и Save позволяют загружать и сохранять BSP-файл в формате ASCII. Нужно просто ввести имя файла, который вы хотите загрузить или сохранить, указав расширение .LEV и щелкнуть на кнопке OK. Команда Exit, естественно, приводит к выходу из системы.
- Options. Меню позволяет выбрать, что именно сделать видимым в окне редактора. Возможно, вы захотите видеть только стены или полы, или выключить сетку. Меню Options позволяет сделать это. В нем содержится три подменю: View Grid (Показать сетку), View Walls (Показать стены) и View Floor (Показать пол). Их можно включать и выключать, щелкнув на них.
- Build. Данное меню состоит из двух команд, но в настоящее время активна только одна из них: Compile BSP and View (Компиляция и отображение BSP). Эта опция осуществляет компиляцию BSP-дерева уровня и переключение в режим трехмерной визуализации. Находясь в трехмерном режиме визуализации, можно использовать клавиши со стрелками для навигации и клавишу <H> для получения подсказки. Клавиша <Esc> позволяет вновь вернуться в режим редактирования и внести новые изменения.
- Help. Это меню имеет единственную команду About (О программе).

#### НА ЗАМЕТКУ

Камера в трехмерном мире помещается в точку (0,0,0), которая помечена зеленым треугольником. Кроме того, ваша высота всегда будет равна текущей высоте стены. Поэтому прежде чем входить в трехмерный мир, убедитесь, что текущая высота стены меньше высоты самой высокой стены, иначе ваша голова будет находиться над потолком :)

Заключительные замечания о редакторе и просмотре уровней. На этом мы закончим описание использования редактора, однако прежде чем переходить к техническому материалу, необходимо упомянуть о некоторых деталях. В большинстве случаев вы будете запускать редактор и начинать с пустого уровня. Если у вас уже есть некий уровень, который вы хотите достроить, он загружается с помощью команды меню File⇒Load .LEV File. После этого в любом случае выбирается высота стены и текстура и добавляются стены. Затем можно перейти в режим рисования пола и добавить несколько плит пола. Когда вы готовы посмотреть на уровень, нужно изменить высоту стены до высоты, на которой вы хотите разместить камеру, и выбрать в главном меню Build⇒Compile BSP and View, чтобы сгенерировать BSP и тестовой визуализации, выйти из которой можно при помощи клавиши <Esc>.

Данный процесс можно продолжать до тех пор, пока уровень не получится таким, как вы хотите. При этом необходимо помнить о следующих вещах. Во-первых, редактор будет генерировать и пол, и потолок из заданных вами плит пола. Потолок будет зеркальным отражением пола, но он визуализируется на высоте самой высокой стены **вашего**

уровня. Кроме того, я советую сохранять свою работу перед попыткой компиляции, поскольку на сложных уровнях система иногда зависает.

## Формат файла редактора BSP

Получив представление о том, как использовать редактор, можно поговорить и о том, как хранятся данные уровня. Файл имеет расширение `.LEV` и представляет собой удобный ASCII-файл, который состоит из заголовка, списка стен и полов. На рис. 13.31 представлен образец файла квадратной комнаты (сам файл носит имя `BSPEXAMPLE01.LEV` и есть на прилагаемом компакт-диске).

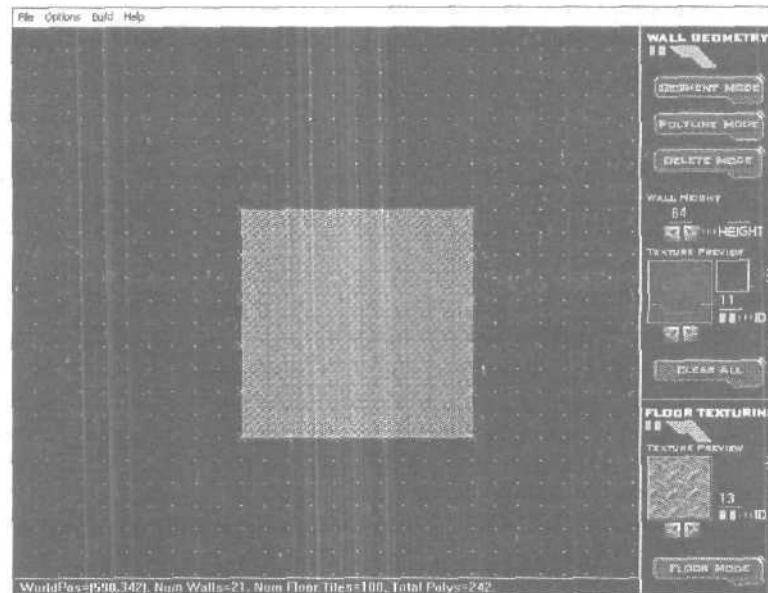


Рис. 13.31. Образец комнаты

```
Version: 1.0
NumSections: 2
Section: walls
NumWalls: 20
240 192 288 192 0 128 11 65535 4169
288 192 336 192 0 128 11 65535 4169
336 192 384 192 0 128 11 65535 4169
384 192 432 192 0 128 11 65535 4169
432 192 480 192 0 128 11 65535 4169
480 192 480 240 0 128 11 65535 4169
480 240 480 288 0 128 11 65535 4169
480 288 480 336 0 128 11 65535 4169
480 336 480 384 0 128 11 65535 4169
480 384 480 432 0 128 11 65535 4169
480 432 432 432 0 128 11 65535 4169
432 432 384 432 0 128 11 65535 4169
384 432 336 432 0 128 11 65535 4169
336 432 288 432 0 128 11 65535 4169
288 432 240 432 0 128 11 65535 4169
240 432 240 384 0 128 11 65535 4169
```



Оно показывает число стен, которые необходимо считать из файла строка за строкой. Каждая стена в настоящий момент состоит из семи значений и имеет следующий вид.

`x0.f y0.f x1.f y1.f elev.f textjd.d color.d attr.d`

Здесь `x0.f`, `y0.f`, `x1.f` и `y1.f` — конечные точки прямолинейного отрезка, `elev.f` и `height.f` — подъем и высота стены (подъем пока что не используется и всегда равен 0), а `textjd.d`, `color.d` и `attr.d` — это идентификатор текстуры, цвет и атрибуты стены.

НА ЗАМЕТКУ

`.f` обозначает числа с плавающей точкой, а `.d` — целые числа.

Например, первая стена из файла `BSPEXAMPLE01.LEV` выглядит следующим образом.

`240 192 288 192 0 128 11 65535 4169`

Она представляет собой прямолинейный отрезок от точки (240,192) до точки (288,192) с подъемом 0.0, высотой 128,0, цветом `RGB(255,255,255)` и атрибутами 4169 (с плоским затенением, двусторонняя, с 16-битовым цветом).

После завершения раздела стен идет ключевое слово `EndSection`, и начинается раздел полов.

Section: floors

Первые две строки определяют размер игрового поля в плитах пола.

`NumFloorsX: 27`

`NumFloorsY: 24`

Плиты пола хранятся как отдельные целые числа, каждое из которых представляет идентификатор текстуры данной плитки. Если некая плитка имеет значение -1, в данном месте нет пола. Значения, большие или равные нулю, указывают индексы текстуры. Программа анализирует их и с учетом положения в матрице генерирует объект типа `OB-JECT4DV2`, состоящий из сетки многоугольников, где каждая плитка поля представлена парой треугольников с соответствующей текстурой.

На этом описание формата файла окончено. Следует отметить, что в него легко можно вносить изменения с помощью любого ASCII-редактора, как можно легко добавить дополнительную функциональность с помощью команд `Section` и `EndSection`, так что это достаточно гибкий формат файла.

## Программаредактора уровня

Редактор уровня является типичной наспех скомпонованной программой и ни в коей мере не может служить образцом программирования. Тем не менее, он вполне справляется со **своими** задачами, и мы будем использовать его, чтобы добавлять новые свойства в будущие демонстрационные программы с использованием BSP (если таковые будут). Редактор уровня содержит определенный набор функций, необходимых для редактирования; эти функции на первый взгляд кажутся простыми, но приходится затратить немало усилий, чтобы заставить их работать. Рассмотрим эти функции.

Первая трудность заключалась в создании графического интерфейса. Мне пришлось писать обработчик кнопок, который позволяет щелкать **мышью**, перетаскивать и рисовать линии в редактируемой области. Проблема в том, что использование смеси меню и других управляющих элементов из DirectX и GDI/Win32 заставило меня оставить обработку мыши операционной системе Windows. Это означает, что события от мыши передаются в функцию `WinProc()`, а не через `DirectInput`, поэтому нужно помещать их в буфер, а затем считывать в цикле `Game_Main()`. Кроме того, поскольку для визуализации дисплея применяется `DirectX`, используется принятая нами частота вывода 30 кадров в секунду.

Иными словами, вывод информации в окне выполняется в реальном времени, так что изображение не является статическим. Это создает определенные проблемы, так что при проектировании редактора необходимо было решать, будет ли дисплей обновляться в реальном времени или нет.

Следующей проблемой является рисование стен и полов. Я использовал прямые линии для стен и масштабированные растровые изображения для представления текстур стен и полов, но разместить все и добиться, чтобы мышь указывала куда следует, всегда непросто. Кроме того, предполагается, что двумерный экран представляет трехмерный вид сверху в плоскости  $xz$  в левой системе координат, поэтому надо корректно выбрать масштаб и **соответствующие** множители для преобразования двумерных данных в трехмерные. Вот и еще одно отчасти произвольное проектное решение наряду с шагом сетки.

Наконец, нам нужна вторичная структура данных, которую можно использовать для представления стен и полов в ходе редактирования, загрузки и сохранения вместо более сложной структуры BSPNODEV1. Я использовал **следующую** структуру.

// Тип bsp, представляющий отдельный двумерный **отрезок**

```
typedef struct BSP2D_LINE_TYP
{
    int id;           // Идентификатор отрезка
    int color;        // Цвет отрезка (многоугольника)
    int attr;         // Атрибуты отрезка (многоугольника)
                    // (режимы затенения и т.п.)
    int texture_id;   // Индекс текстуры
    POINT2D p0, p1;   // Конечные точки линии

    int elev, height; // Высота стены
} BSP2D_LINE, *BSP2D_LINE_PTR;
```

Эта структура используется только в пределах редактора и не имеет никакого отношения к трехмерному миру; она просто помогает представить двумерный мир редактора. Плиты пола в данном случае просто двумерный массив целых чисел.

```
int floors[BSP_CELLS_Y-1][BSP_CELLS_X-1];
```

Когда выполняется команда меню Build  $\Rightarrow$  Compile BSP and View, стены преобразуются из формата BSP2D\_LINE в связанный список узлов BSPNODEV1. Это осуществляется с помощью функции Convert\_Lines\_To\_Walls(), которая слишком велика, чтобы приводить ее здесь; данная функция преобразует список двумерных стен в трехмерное множество многоугольников и сохраняет их в виде единого связанного списка узлов BSPNODEV1. Затем корень передается функции Bsp\_Build\_Tree(), которая строит дерево.

Остались полы и потолки. Пол представляет собой не более чем сетку, которую предстоит сгенерировать. Поскольку потолок — это просто перемещенный пол, можно использовать тот же самый тип OBJECT4DV2. Полы генерируются следующей функцией.

```
int Generate_Floors_OBJECT4DV2(
    OBJECT4DV2_PTR obj, // Указатель объекта
    int rgbcolor,        // Цвет пола без текстуры
    VECTOR4D_PTR pos,   // Начальное положение
    VECTOR4D_PTR rot     // Начальный поворот
    int poly_attr;       // Атрибуты затенения
```

Функции передается указатель выводимого объекта, цвет пола под текстурой (обычно белый), положение, угол поворота, атрибуты многоугольников, которые должна иметь

сетка пола, после чего функция генерирует пол. Пример генерации пола можно увидеть на рис. 13.32.

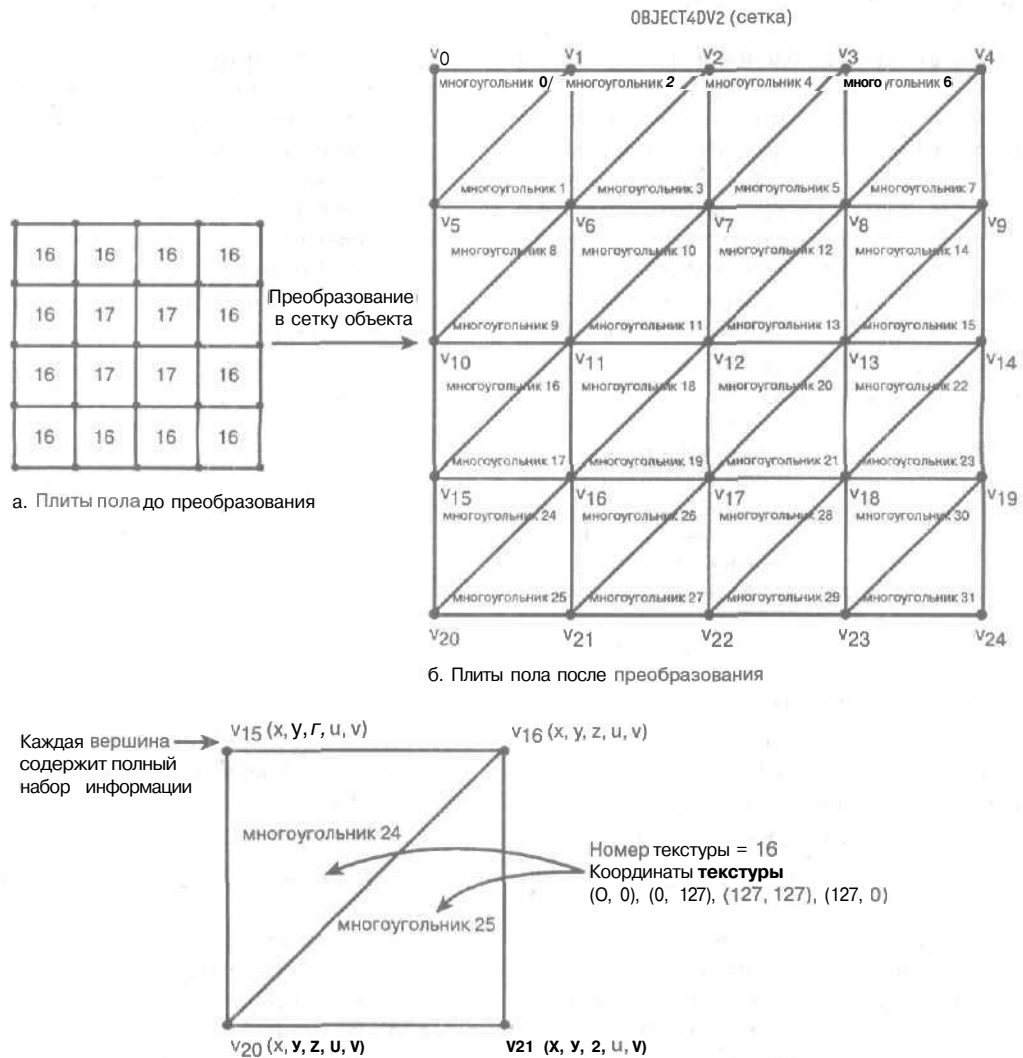


Рис. 13.32. Преобразование пола в сетку

После того как будет сгенерирован пол, редактор переключается в трехмерный демонстрационный режим, с которым мы уже имели дело. Единственное отличие в данном случае заключается в том, что в нем не используется сортировка или Z-буферизация, а вставка геометрических объектов в список визуализации для каждого кадра всегда осуществляется в одном и том же порядке:

1. полы;
2. потолки;
3. BSP-дерево.

Тем самым гарантируется корректная упорядоченность, поскольку полы и потолки всегда находятся "позади" BSP, а BSP-деревья сами по себе обеспечивают правильный порядок содержащихся в них объектов.

## Недостатки BSP

Хотя BSP-дерево, несомненно, позволяет быстро определить порядок визуализации многоугольников в процессе выполнения, этот метод не лишен недостатков. Во-первых, пространство должно быть квазистатическим. Это означает, что многоугольники нельзя вращать, но в некоторых случаях их можно передвигать. Например, рассмотрим рис. 13.33. На рисунке представлено изображение двух стен. Если подвинуть одну из стен вперед, как показано на рис. 13.33б, их порядок останется прежним. Таким образом, BSP-дерево можно использовать, когда допустимы сдвиги многоугольника в определяемой им плоскости, но не более того.

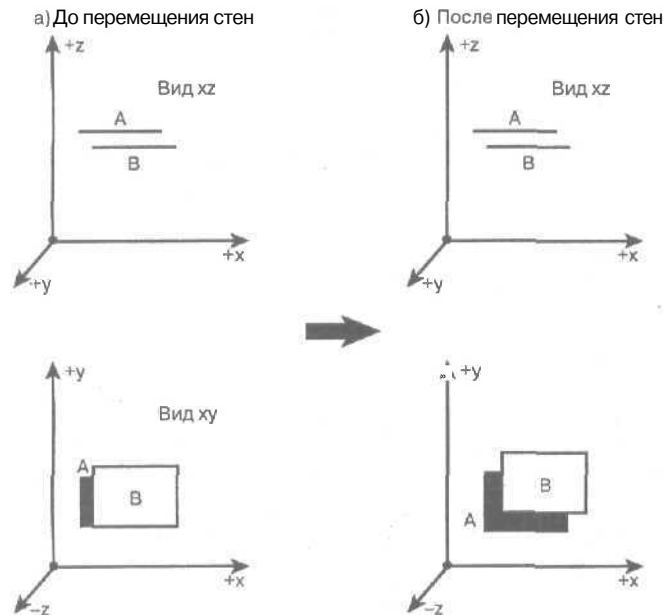


Рис. 13.33. BSP-дерево инвариантно при перемещении стен вдоль оси  $z$

Вторым существенным ограничением BSP-деревьев является то, что они, очевидно, непригодны для имитаторов или игр с большим количеством движений, поскольку в этом случае в каждом кадре придется генерировать BSP-дерево заново. Тем не менее, BSP-дерево является очень красивым решением проблемы визуализации и может использоваться в ситуациях, когда игрок движется в практически статической среде. Кроме того, назначение BSP отнюдь не исчерпывается определением порядка визуализации.

## Стратегии с нулевым перерисовыванием на основе BSP

BSP-деревья можно использовать для определения порядка следования многоугольников от дальних к ближним в сцене с заданным множеством статических многоугольников для любой точки наблюдения. Этот порядок определяется путем обхода BSP-дерева с помощью модифицированного симметричного рекурсивного поиска и посещения узлов в порядке от дальних к ближним, как это происходит в алгоритме художника.

Однако при этом получается, что во многих случаях один и тот же пиксель будет выводиться снова и снова. Это большая проблема в любом алгоритме сортировки или Z-буферизации. Правильный порядок визуализации определен, но один и тот же пиксель может перерисовываться много (иногда сотни) раз.

Рассмотрим, например, рис. 13.34, на котором представлено множество многоугольников в плоскости  $xz$ . Многоугольники упорядочены от дальних к ближним (или визуализированы с помощью Z-буфера). При использовании Z-сортировки каждый пиксель, составляющий наименьший многоугольник данного множества, будет перезаписан остальными многоугольниками. Каждый пиксель следующего по величине многоугольника будет перезаписан всеми большими многоугольниками и т.д. В этом простом случае каждый пиксель каждого удаленного многоугольника будет рисоваться от 1 до  $(p-1)$  раз, где  $p$  — число многоугольников в данном множестве.

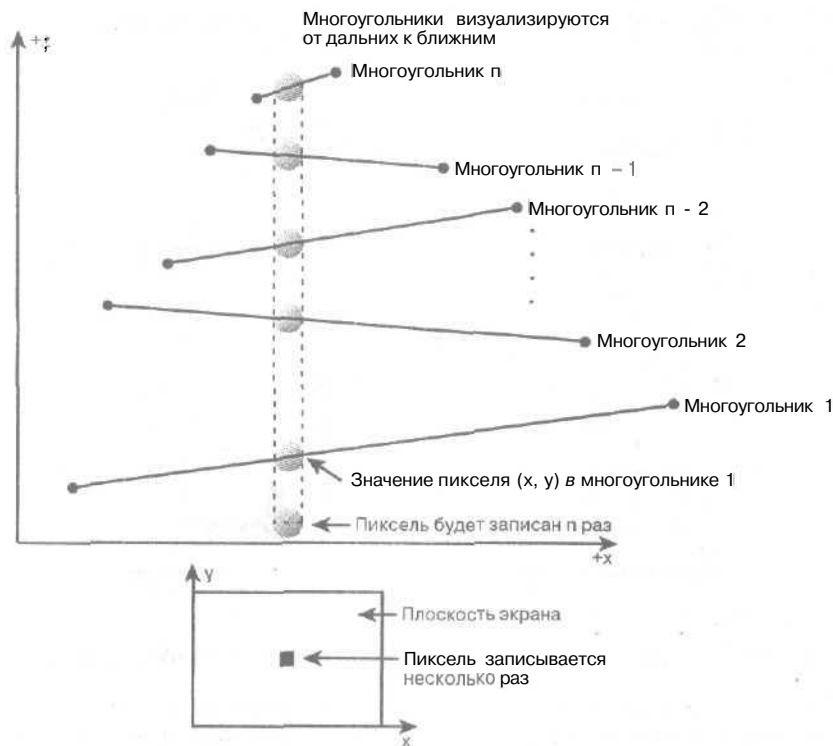


Рис. 13.34. Наихудший случай наложения

Версия с Z-буферизацией точно так же плоха. В этом случае приходится не только перезаписывать пиксели (если значения  $z$  ближе), но еще и постоянно проверять Z-буфер. Поэтому, хотя перезаписей в общем случае не так много, это достигается за счет огромного количества операций считывания/записи Z-буфера. Время теряется если не на рисование пикселей, то на тестирование пикселей, которые не будут видны. Вывод: необходимо искать решение проблемы наложения.

Помочь в этом могут BSP-деревья. Вместо того, чтобы обходить BSP-дерево в порядке от более удаленных объектов к менее удаленным, можно совершить обход в порядке от менее к более удаленным. Тогда при посещении каждого многоугольника маскируется запись

данных на экран, т.е. многоугольники используются как своего рода буфер трафаретов. Теоретически это хорошо, но в действительности мы **ВНОВЬ** оказываемся в той же ситуации, что и с Z-буферизацией (проверка некоего условия и запись пикселя в случае его выполнения). Нам удастся избежать наложения, но все равно придется выполнять проверку для каждого отдельного пикселя. Можно ли более разумно использовать преимущества порядка от менее удаленных к более удаленным объектам? Ответ — да, можно. Одним из способов решения данной проблемы является использование построчной визуализации.

Построчная визуализация работает **следующим** образом. Вместо того, чтобы рисовать многоугольник пиксель за пикселем, для каждой строки развертки вычисляются только конечные точки, которые затем помещаются в связанный список или структуру данных. По существу, мы блокируем внутренний цикл, который выполняет растеризацию каждой строки развертки, и вместо этого помещаем в список данные о конечных точках. Затем для каждого многоугольника выполняется сортировка списка строк развертки по  $x$  и  $y$ , таким образом, получается набор блоков строк развертки для каждой строки экрана. Итак, нам удалось **разделить** задачу на коллекции строк развертки. Данный процесс показан на рис. 13.35. Если мы сможем решить задачу для отдельной строки развертки, то сможем решить и задачу в целом.



**Рис. 13.35.** *Сохранение многоугольников в виде строк развертки вместо растеризации "на лету"*

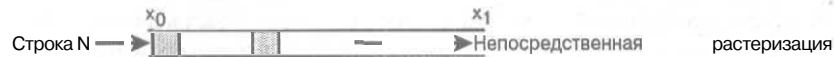
Рассмотрим, что происходит в различных случаях, представленных на рис. 13.36.

- В случае 1 в строке  $p$  содержится одна строка развертки. В таком случае эта линия развертки растеризуется, и задача решена.
- В случае 2 в строке  $p$  содержится две строки развертки, но они не перекрываются по  $x$ . В этом случае просто растеризуется каждая из них, поскольку они не могут перекрываться.
- В случае 3 за одной строкой развертки включена другая, но вторая строка развертки полностью закрыта первой, поэтому вторая линия развертки полностью игнорируется и наложения не происходит.
- В случае 4 за одной строкой развертки следует другая, но теперь вторая строка развертки лишь частично закрыта первой. В этом случае отсекается часть второй строки развертки, закрытая первой, и рисуется только оставшаяся часть второй строки — и снова никакого наложения не происходит.

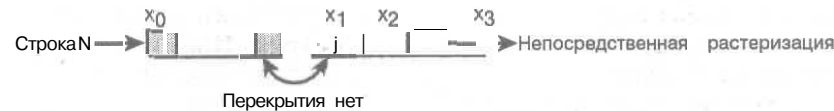
Эти случаи показывают, какие варианты ситуаций могут возникнуть, и как их следует **обрабатывать**. Используя технологию строк развертки совместно с визуализацией от ближних

объектов к дальним, можно добиться совершенной визуализации с нулевым наложением. Однако у всего есть своя цена: в данном случае это — время, необходимое, чтобы определить, является ли некая строка развертки полностью скрытой или *усеченной*, и выполнить объединение строк развертки. Заставить такой алгоритм работать на практике (с координатами текстыры и т.д.) — непростая задача. Тем не менее, даже приблизительная *версия*, не удаляющая наложение на все 100%, все равно стоит затраченных на нее усилий.

Случай 1. Единственная строка развертки включена в буфер строки N



Случай 2. Две линии развертки включены в буфер строки N



Случай 3. Включены две строки развертки, но вторая заслонена первой



Случай 4. Включены две линии развертки с частичным перекрытием



Рис. 13.36. Возможные случаи расположения интервалов строки развертки

## Использование BSP-деревьев для отбраковки

Одним из важнейших свойств любого алгоритма разбиения пространства является способность быстро отбрасывать большое количество информации о многоугольниках из конвейера *визуализации*. Сегодня зачастую именно это свойство (наряду с определением столкновений) является единственным мотивом использования BSP-деревьев. Свойства образуемых BSP-деревьями выпуклых подпространств, а также линейное время обхода делают их идеальными для широкомасштабной отбраковки. Мы рассмотрим два различных метода отбраковки на основе BSP-деревьев: *отбраковку задних поверхностей* (back-face culling) и *пространственную отбраковку* (frustum culling). Первый метод в большей степени низкоуровневый, второй же — более высокоуровневый, который действительно *позволяет* использовать способность структуры BSP отбрасывать многоугольники, которые не являются потенциально видимыми.

### Отбраковка задних поверхностей

Даже в процессе простого обхода BSP с целью включения многоугольников в список визуализации можно осуществлять отбор невидимых поверхностей или отсечение составляющих BSP многоугольников, используя для этого *функцию* вставки. Однако в большинстве случаев стены являются двусторонними и стандартная операция удаления

задней поверхности оказывается неэффективна — поскольку все грани будут проходить устройство отбраковки без потерь. Следовательно, чтобы удалить невидимые поверхности в ходе сканирования при включении многоугольников в список, необходимо применить более совершенный алгоритм отбраковки.

Рассмотрим постановку задачи удаления невидимых поверхностей, приведенную на рис. 13.37. Как показано на рисунке, сначала вычисляется (или извлекается, если она была предварительно вычислена) нормаль к поверхности многоугольника, а затем находится скалярное произведение вектора нормали и вектора к точке наблюдения. Если угол между векторами больше  $90^\circ$  (скалярное произведение  $< 0$ ), многоугольник является задней, невидимой поверхностью. Однако такой подход работает только для односторонних многоугольников,

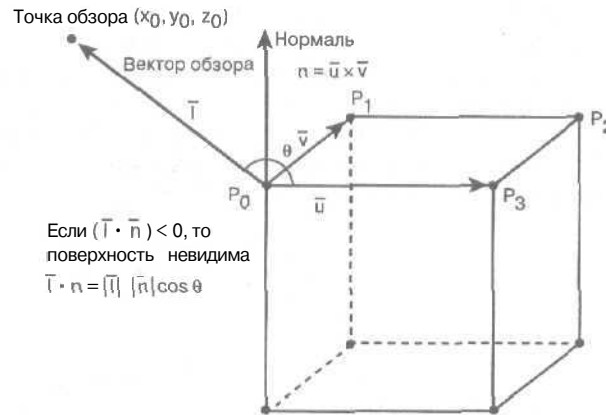


Рис. 13.37. Подробный анализ процесса удаления невидимой поверхности

Кроме того, алгоритм удаления невидимых поверхностей не принимает во внимание вектор направления взгляда. Например, на рис. 13.38 угол между нормалью к многоугольнику и вектором к точке наблюдения меньше  $90^\circ$ , но если учесть вектор направления взгляда, окажется, что даже когда стандартный алгоритм удаления невидимых поверхностей не удаляет данный многоугольник, его фактически невозможно увидеть.



Рис. 13.38. В алгоритме удаления невидимых поверхностей не учитывается направление наблюдения

Если удаление невидимых поверхностей выполняется в системе координат камеры, подобные ситуации не возникают, поскольку в этом случае вектор **взгляда** всегда направ-

лен вдоль положительной оси  $z$   $(0,0,1)$ . Следовательно, алгоритм удаления невидимых поверхностей в этом случае работает правильно и не отправляет лишние многоугольники в конвейер визуализации. Возникает вопрос, стоит ли преобразовывать многоугольники в координаты камеры только для того, чтобы их удалить? Конечно, не стоит. Поэтому мы выполняем отбор невидимых поверхностей в мировых координатах, в результате лишние многоугольники оказываются позади точки наблюдения и тривиальным образом удаляются на стадии отсекания.

С учетом приведенной выше информации, для BSP-дерева, в котором многоугольники в основном являются двусторонними, нам необходим более строгий метод отбраковки, если мы хотим, чтобы он работал на стадии обхода и включения BSP в список визуализации. Можно, конечно, игнорировать всю эту оптимизацию, отправив в конвейер весь список визуализации, и пусть его обрабатывает обычная функция удаления невидимых поверхностей. Но дело в том, что это весьма удачная оптимизация, которая очень пригодится нам в следующем разделе при отборе целых подпространств.

Рассмотрим, что нужно **знать**, чтобы решить задачу удаления невидимых поверхностей для двусторонних многоугольников в мировом пространстве, учитывая как положение точки наблюдения, так и направление обзора. Постановка задачи представлена на рис. 13.39.

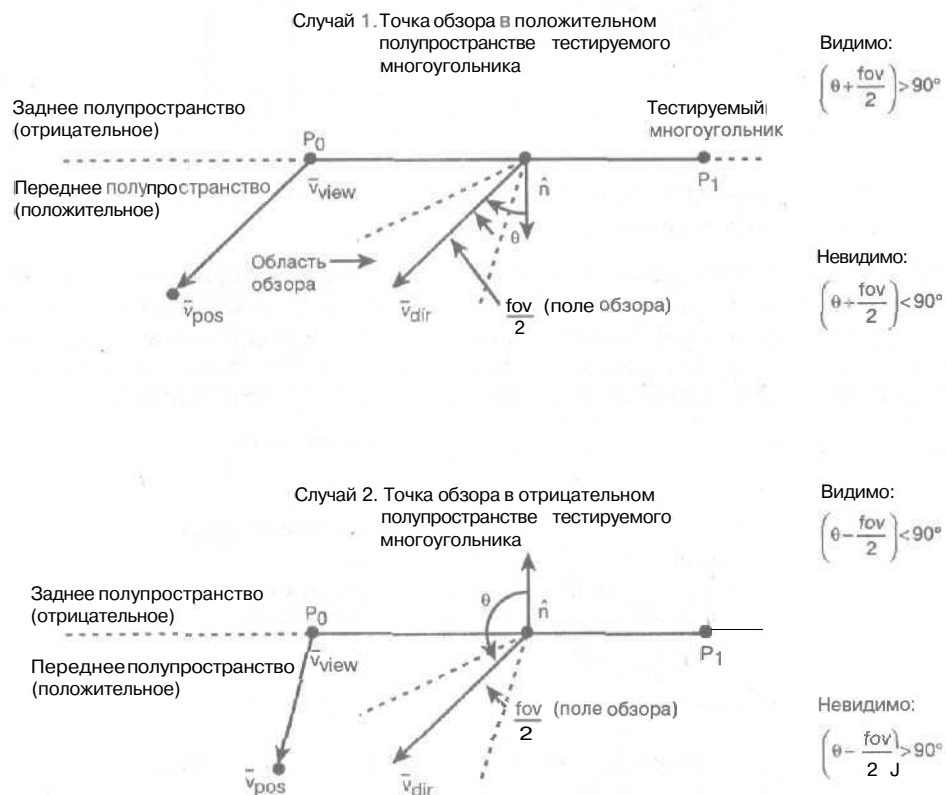


Рис. 13.39. Постановка задачи полного удаления невидимых поверхностей с учетом положения наблюдателя и направления обзора

Прежде всего необходимо определить, с какой по отношению к исследуемому многоугольнику стороны находится точка наблюдения. Затем нужно вычислить угол между нормалью к поверхности исследуемого многоугольника и вектором обзора. Однако нужно быть внимательным при учете поля зрения области видимости. Иными словами, даже если угол между положительной нормалью к поверхности и вектором обзора (не путать с вектором от многоугольника к точке наблюдения) больше  $90^\circ$ , многоугольник все еще может находиться в поле зрения. Это происходит потому, что поле зрения расширяет допустимый диапазон углов, в котором исследуемый многоугольник может быть видимым, как показано на рис. 13.39.

Таким образом, при выполнении вычислений необходимо учитывать поле зрения. Первым делом нужно вычислить (или использовать вычисленную ранее) нормаль к поверхности исследуемого многоугольника и найти скалярное произведение этой нормали и вектора обзора, чтобы определить, с какой стороны находится точка наблюдения по отношению к лицевой/обратной ориентации многоугольника.

Определив, с какой стороны находится точка наблюдения, необходимо найти второе скалярное произведение — нормали к той поверхности многоугольника, с которой находится точка наблюдения, и вектора направления взгляда. Этот результат является решающим фактором. Существует два возможных случая. Если точка наблюдения находится с лицевой стороны исследуемого многоугольника и  $(\theta + \text{FOV}/2) > 90^\circ$ , то многоугольник видим, в противном случае он невидим. Если же точка наблюдения находится с обратной стороны многоугольника, то для того, чтобы узнать, будет ли он видимым, нужно проверить условие  $(\theta - \text{FOV}/2) < 90^\circ$ . Математически это выглядит следующим образом.

Дано:

- $\mathbf{v}_{\text{dir}}$  — вектор направления взгляда;
- $\mathbf{v}_{\text{pos}}$  — положение точки наблюдения;
- $\mathbf{n}$  — нормаль к поверхности;
- $\mathbf{v}_{\text{view}}$  — вектор от поверхности к точке наблюдения;
- $\theta$  — угол между нормалью к поверхности  $\mathbf{n}$  и вектором направления взгляда  $\mathbf{v}_{\text{dir}}$ .

Случай 1. Точка наблюдения  $\mathbf{v}_{\text{pos}}$  находится с лицевой стороны исследуемого многоугольника, т.е.  $(\mathbf{n} \cdot \mathbf{v}_{\text{view}}) > 0$ . Тогда условие видимости  $(\theta + \text{FOV}/2) > 90^\circ$ . Однако  $\mathbf{n} \cdot \mathbf{v}_{\text{dir}} = |\mathbf{n}| |\mathbf{v}_{\text{dir}}| \cos \theta$ , поэтому, считая векторы нормализованными ( $|\mathbf{n}| = |\mathbf{v}_{\text{dir}}| = 1$ ), получим  $\mathbf{n} \cdot \mathbf{v}_{\text{dir}} = \cos \theta$  и  $\theta = \arccos(\mathbf{n} \cdot \mathbf{v}_{\text{dir}})$ . Таким образом, можно просто вычислить значение  $\theta$  и подставить его в условие видимости или использовать преобразование, чтобы определить, подлежит ли отбраковке рассматриваемый многоугольник. К сожалению, в реальном времени невозможно использовать функцию  $\arccos()$ , но мы вернемся к этому чуть позже, а пока проанализируем второй случай расположения точки наблюдения.

Случай 2. Точка наблюдения  $\mathbf{v}_{\text{pos}}$  находится с обратной стороны исследуемого многоугольника, т.е.  $\mathbf{n} \cdot \mathbf{v}_{\text{view}} < 0$ . Тогда условие видимости имеет вид  $(\theta - \text{FOV}/2) < 90^\circ$ . Угол  $\theta$  вычисляется так же, как и в случае 1 и подставляется в условие видимости для обратной стороны.

### Быстрое вычисление арккосинуса

К сожалению, при работе в реальном времени невозможно использовать математическую функцию для вычисления арккосинуса. Нужно применить еще один специальный прием, а именно — таблицу поиска. Но сначала обсудим, что же именно делает функция  $\arccos$ .

Это функция, обратная косинусу, иными словами, для заданного значения  $x \in [-1,1]$  она находит значение  $y$  такое, что  $\cos(y) = x$ . Функция  $y(x)$  в таком случае называется функцией, обратной косинусу, или арккосинусом. Например,  
 $\cos(30^\circ) = 0.866$ ,  
 $\arccos(0.866) = 30^\circ$ .  
 Объединяя данные функции, получаем:  
 $\cos(\arccos(0.866)) = 0.866$ .

НА ЗАМЕТКУ

Углы в примерах указаны в градусах.

Для того чтобы достаточно быстро вычислять арккосинус, можно создать таблицу поиска. Необходимо отобразить значения  $[-1,1]$  на углы  $[0,180^\circ]$ , используя арккосинус в качестве функции отображения. Мы не можем использовать таблицу с отрицательными или десятичными числами, поэтому сначала сместим диапазон, прибавив 1 к значению  $x$ , что даст нам вместо интервала  $[-1,1]$  интервал  $[0,2]$ . Увы, таблица поиска, содержащая всего три элемента, для нас по сути бесполезна, поэтому надо заодно изменить масштаб диапазона, чтобы таблица содержала, по крайней мере, одно значение для каждого возможного угла в диапазоне  $[0,180^\circ]$ . Для этого умножим все значения на 90, получая желаемый диапазон  $90 \cdot [0,2] = [0,180]$ .

Итак, функция отображения выглядит следующим образом:

Область определения  $x \in [-1,1]$ .

Функция отображения  $f(x) = (x + 1) \cdot 90 = \text{index}$ .

Теперь мы можем использовать значение  $\text{index}$  в качестве индекса в таблице поиска, которая содержит арккосинусы для 181 значения, причем значения эти равномерно распределены в интервале  $[-1,1]$  и отображены на диапазон  $[0,180]$ . Такая таблица показана на рис. 13.40.

Таблица поиска отображает значения  $[-1,1]$  на углы  $0-180^\circ$  при помощи функции  $\arccos$

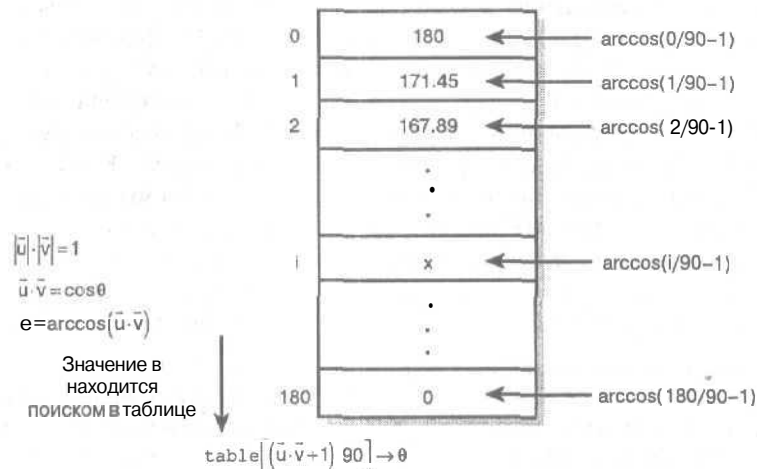


Рис. 13.40. Таблица поиска для вычисления обратного косинуса

Если нам нужна более высокая точность, можно увеличить масштаб отображения, т.е. разбить интервал  $[-1,1]$  на 360 частей, представляющих углы от 0 до  $180^\circ$ , с шагом в полградуса. Ниже представлена программа, которая реализует все сказанное. Сначала мы создаем хранилище для таблицы поиска.

```
float dp_inverse_cos[360+2]; // от 0 до 180 с шагом 0.5
// градусов; +2 для заполнения
```

Таким образом, определяется глобальная таблица поиска для 360 элементов (+2 добавлено для того, чтобы не возникало переполнение, когда доступ к 360 или 361 элементам производится в ходе отображения с небольшим переполнением разрядов после запятой). Чтобы упростить программу, был создан следующий макрос для вычисления арккосинуса  $x$ , где  $x$  принимает значения в диапазоне  $[-1,1]$ .

```
#define FAST_INV_COS(x) \
(dp_inverse_cos[(int)((float)x+1)*(float)180]))
```

Далее приводится код функции, которая генерирует таблицу поиска.

```
void Build_Inverse_Cos_Table(
    float *invcos, // Область памяти для таблицы
    int range_scale) // Диапазон таблицы
{
    // Данная функция строит таблицу арккосинусов,
    // используемую для поиска угла по значению скалярного
    // произведения. Функция отображает значения  $[-1, 1]$  в
    // интервал  $[0, range]$  и разбивает данный интервал на  $p$ 
    // интервалов, при этом размер интервала равен
    //  $180/range$ . Затем функция вычисляет арккосинус для
    // каждого значения от -1 до 1, масштабированного и
    // отображенного, как описано выше, и помещает
    // полученные значения в таблицу. Например, если значение
    // range равно 360, то интервал будет составлять
    //  $[0,360]$ , следовательно, поскольку арккосинус всегда
    // находится в пределах от 0 до 180, каждый элемент
    // массива будет представлять приращение 0.5 градуса.
    // Таблица должна быть достаточно большой, чтобы
    // вместить результаты, объем которых
    //  $=(range\_scale+1)*sizeof(float)$ 

    // Чтобы использовать таблицу, необходимо масштабировать
    // диапазон значений  $[-1, 1]$  к размеру таблицы,
    // например, если строится таблица с диапазоном 0..180,
    // то для доступа к значениям из интервала  $x = [-1, 1]$ 
    // надо найти  $invcos[(x+1)*180]$ ; результатом будет угол
    // в градусах с точностью до 1 градуса

    float val = -1; // Начальное значение

    // Создание таблицы
    for (int index = 0; index <= range_scale; index++)
    {
        // Включение очередного элемента в таблицу (в
        // градусах)
```

```

val = (val > 1) ? 1 : val;
invcos[index] = RAD_TO_DEG(acos(val));

// Приращение значения
val += ((float)1/(float)(range_scale/2));

} // forindex

// Вставка одного дополнительного элемента с тем, чтобы
// если при доступе к таблице произошло небольшое
// превышение значения 1, то это не привело бы к выходу
// за пределы области значений, и мы могли остаться в
// рамках логики вычислений с плавающей запятой
invcos[index] = invcos[index-1];

} // Build_Inverse_Cos_Table

```

В разделе инициализации вашей игры производится вызов функции `Build_Inverse_Cos_Table()` со следующими параметрами.

```

// Построение таблицы поиска для вычисления арккосинуса
Build_Inverse_Cos_Table(dp_inverse_cos, 360);

```

Тем самым мы сообщаем функции, что она должна использовать в качестве хранилища таблицы массив `dp_inverse_cos`. Таблица будет содержать 360 элементов, представляющих диапазон 0-180 с точностью 0.5 градуса.

Для вычисления арккосинуса можно использовать макрос `FAST_INV_COS`, который предполагает наличие таблицы с 362 элементами.

Теперь, при наличии возможности быстрого вычисления арккосинуса, мы можем переписать функцию вставки BSP в список визуализации, которая осуществляет отбраковку как с учетом направления обзора, так и с учетом положения точки наблюдения.

#### НА ЗАМЕТКУ

И все же, почему мы используем 362 элемента? Нам нужны элементы от 0 до 360, т.е. всего 361 элемент, но работу ряда алгоритмов можно ускорить, если игнорировать возможное округление числа с плавающей точкой до 361 вследствие погрешности вычислений. Обычно 361-й элемент является просто копией 360-го.

### Ориентация вектора обзора

Прежде чем написать новую функцию вставки BSP, осталось решить последнюю проблему. Поскольку мы не выполняем преобразование мировых координат в координаты камеры, нам неизвестно направление вектора обзора. Обычно мы используем углы Эйлера или UVN-представление для преобразования мировых координат, а затем предполагаем, что камера находится в точке  $(0,0,0)$  и ориентирована вдоль положительной оси  $z$ ; иными словами, вектор направления обзора —  $(0,0,1)$ .

Вектор направления обзора достаточно легко вычислить на основе кадров: мы просто преобразуем вектор  $(0,0,1)$  с помощью матрицы преобразования в координаты камеры (точнее, обратной к ней) и получаем вектор направления обзора. Однако нам не нужно выполнять все преобразование целиком, включая перенос вектора — надо только повернуть вектор  $(0,0,1)$  на соответствующие углы. Эти углы для модели Эйлера равны `cam.dir.x`, `cam.dir.y` и `cam.dir.z`. Естественно, важен порядок вращения, но для построения матрицы преобразования к координатам камеры мы использовали следующий вызов.

```

Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_XYZ);

```

Это означает выполнение поворота вектора в порядке XYZ (точнее, обратном к нему). Иными словами, производится **вращение** вектора (0,0,1) с помощью **матрицы** вращения XYZ с углами `cam.dir.x`, `cam.dir.y` и `cam.dir.z`. Это можно осуществить следующим образом.

```
MATRIX4X4 mrot;
```

```
Build_XYZ_Rotation_MATRIX4X4(cam.dir.x,
                             cam.dir.y,
                             cam.dir.z
                             &mrot);
```

Затем с помощью данной матрицы мы преобразуем вектор направления обзора.

```
VECTOR4D vz = {0,0,1,1}; // z - 1, w - 1
```

```
Mat_Mul_VECTOR4D_4X4(&vz,
                     &mrot,
                     &vdir);
```

Теперь у нас есть все, что нужно. Может показаться, что мы потратили слишком много усилий, чтобы вычислить векторы и удалить невидимые поверхности, в то время как алгоритм удаления невидимых поверхностей в ходе визуализации может дать практически те же результаты. Однако время, потраченное на выполнение проверок, меньше, чем время, потраченное на преобразование каждого включаемого в список визуализации четырехугольника в два треугольника, не говоря уж о времени на преобразование их из мировых координат в координаты камеры. Напомним, что, пока мы работаем с **BSP-деревом**, все стены остаются четырехугольниками, которые еще не надо разбивать на два треугольника, поэтому, если есть такая возможность, следует использовать BSP.

В заключение отметим, что, хотя этот раздел важен и сам по себе, он является своего рода подготовительным материалом к следующему, где речь пойдет о пространственном отборе, который позволит нам удалять сразу целые поддеревья BSP. Далее **представлена** новая функция, а **точнее** — ее прототип, которая обладает дополнительной возможностью удалять невидимые поверхности в процессе вставки BSP в список визуализации.

```
void Bsp_Insertion_Traversal_RemoveBF_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rend_list, // Список визуализации
    BSPNODEV1_PTR root           // Корень включаемого
                                // BSP-дерева
    CAM4DV1_PTR cam,             // Камера
    int insert_local=0);          // флаги вставки
```

Соглашения о вызове для данной функции точно такие же, как и для предыдущей версии, не поддерживающей удаление невидимых поверхностей. Поэтому данные функции можно **безопасно** менять местами. Программный код, осуществляющий удаление невидимых поверхностей, идентичен тому, который приводился несколькими страницами ранее, поэтому для экономии места листинг здесь не приводится, однако он, как и все другие материалы данной главы, есть на прилагаемом к книге компакт-диске.

В качестве демонстрации работы данной функции рассмотрим программу **DEMOII13\_2.CPP|EXE**. Это стандартный редактор уровня, но при переходе в трехмерный режим на экран внизу выводится число многоугольников, передаваемых в конвейер визуализации функцией вставки BSP. Используя клавишу <C>, можно переключиться с использования стандартной функции вставки BSP на использование новой версии и пронаблюдать, как уменьшается количество многоугольников, поступающих в список визуализации, **при** выборе различных точек наблюдения.

Чтобы самостоятельно скомпилировать демонстрационную программу, необходим файл DEMOII13\_2.CPP, а также библиотечные модули T3DLIB1-11.CPP и библиотечные файлы DirectX. Кроме того, вам понадобится файл ресурсов для меню и т.п.; имя этого файла — DEMOII13\_2.RC. Для экспериментов воспользуйтесь файлом DOOM01.LEV.

При запуске демонстрационной программы вас может удивить, что, находясь вне помещения, вы все еще видите многоугольники, включенные в список визуализации из BSP-дерева. В принципе это правильно, поскольку мы осуществляем отбор плоскостей, а не конечных многоугольников. Если мы хотим выйти на следующий уровень отбраковки, то придется проверять не только то, видима ли плоскость, но и попадают ли конечные точки многоугольника в область видимости. Это гораздо сложнее и приводит к уменьшению отдачи. Однако это можно попробовать сделать так, как показано на рис. 13.41.

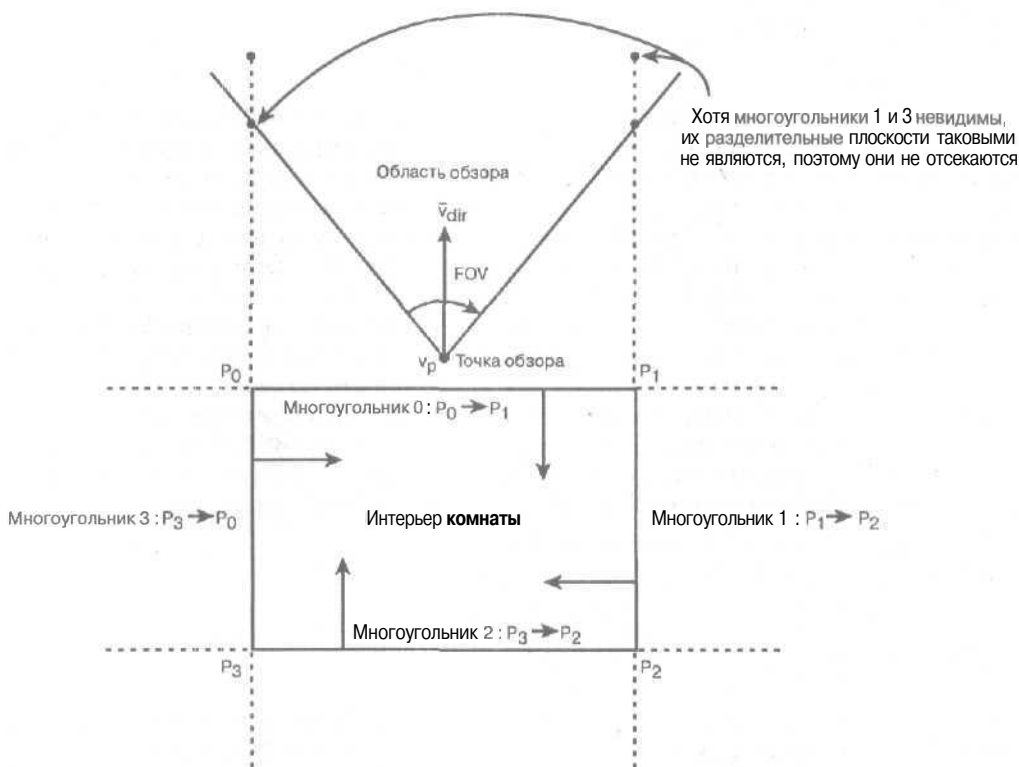


Рис. 13.41. Многоугольники могут быть невидимы, но мы удаляем разделительные плоскости целиком

## Пространственная отбраковка

В BSP-деревьях можно использовать более мощную операцию отбраковки, чем тривиальное удаление невидимых поверхностей. В принципе мы уже начали писать ее, когда разрабатывали функцию отбраковки невидимых поверхностей, но так и не закончили. Рассмотрим рис. 13.42. Точка наблюдения находится в отрицательном полупространстве плоскости разделения. Учитывая направление взгляда, очевидно, что плоскость разделения, невидима, но более важным является то, что невидимо также все положительное полупространство данной плоскости разделения!

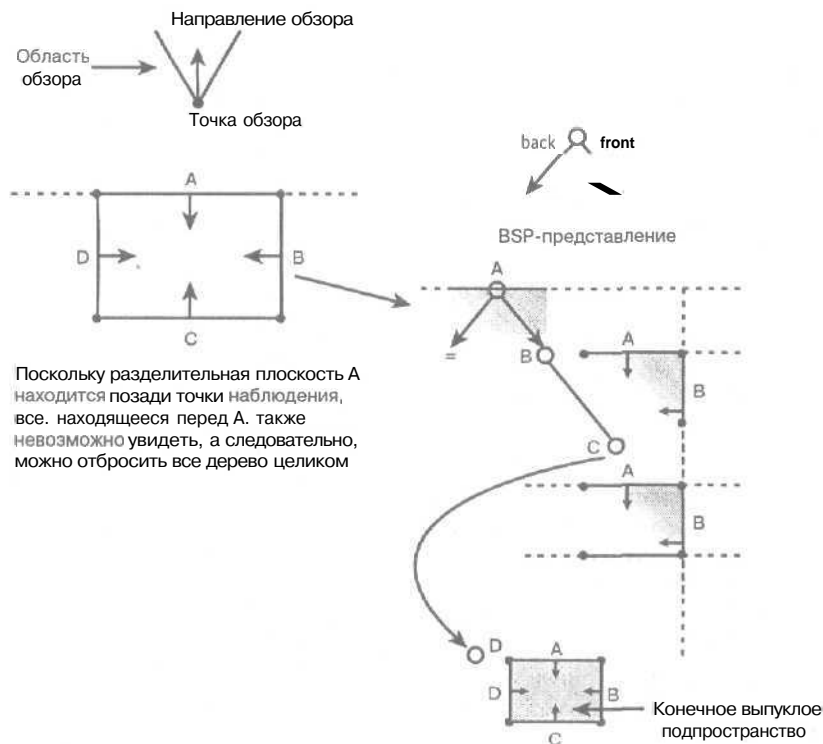


Рис. 13.42. Пространственная отбраковка

Иными словами, можно отбросить целое поддерево, **исходящее** из **текущего** узла, и не рассматривать его вовсе! Подумайте, насколько это мощный метод: найдя такую плоскость разделения, для которой точка наблюдения находится перед ней (или позади нее), и данная плоскость при имеющемся направлении обзора невидима, можно отбросить не только саму эту плоскость, но и все поддерево целиком. Это замечательное свойство **BSP-дерева**. Оно является следствием того, что BSP представляет пространство в виде выпуклых подпространств.

Для большей убедительности рассмотрим рис. 13.43. Здесь представлен простой пример из двух комнат (предполагается, что **BSP-дерево** уже создано). Точка наблюдения находится в комнате 1, а стена позади игрока и все содержимое комнаты 2 являются невидимыми. Мы можем сразу определить это, заметив, что стена 3 является невидимой, следовательно, все подпространство за ней также является невидимым. Таким образом, совершать дальнейший обход нет необходимости.

Реализация пространственной отбраковки на удивление **проста**, и мы уже частично написали ее. Теперь можно или написать отдельную функцию, которая помечает отбрасываемые узлы со всеми их дочерними узлами, или можно делать это в процессе выполнения, как и при удалении невидимых поверхностей. Поскольку нет причин обходить дерево дважды, попытаемся встроить возможность совершать отбраковку в функцию вставки BSP, как мы уже проделали это с удалением невидимых граней. Идея заключается во включении рекурсивных вызовов по посещению передних или задних деревьев в операции отбраковки. Иными словами, после того, как мы выясним, что некий многоугольник отбрасывается, мы знаем, что все позади него также отбрасывается и, таким образом, нет необходимости выполнять рекурсивные вызовы и обходить заднее поддерево данного узла. Это единственное изменение, **требующее** одной строки кода.

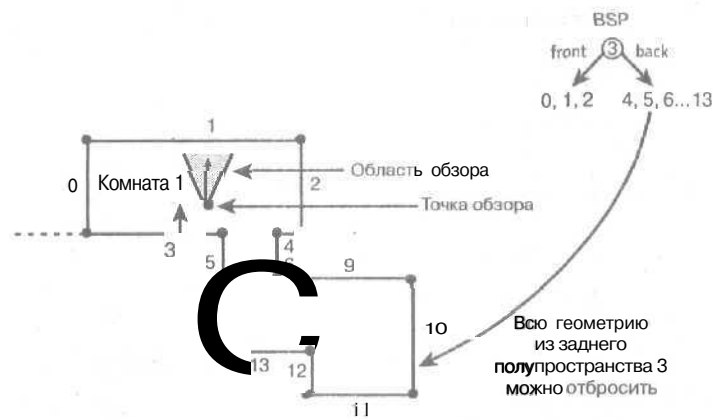


Рис. 13.43. Пространственная отбраковка целых комнат

Новая функция, реализующая данный алгоритм, имеет следующий прототип.

```
void Bsp_Insertion_Traversal_FrustumCull_RENDERLIST4DV2(
    RENDERLIST4DV2_PTR rendlist
    BSPNODEV1_PTR root
    CAM4DV1_PTR cam,
    int insert_local=0);
```

Эта функция **вызывается** аналогично двум предыдущим, единственное отличие заключается в том, что она выполняет как удаление невидимых поверхностей, так и пространственную отбраковку.

Чтобы посмотреть ее в действии, воспользуйтесь демонстрационной программой DEMOII13\_3.CPP|EXE. Это стандартный редактор уровня, который при переходе в **трехмерный** режим выводит на экран количество многоугольников, передаваемых в конвейер визуализации функцией вставки BSP. Используя клавишу <C>, можно переключиться с использования стандартной функции вставки BSP на использование новой версии и пронаблюдать, как уменьшается количество многоугольников, поступающих в список визуализации при выборе различных точек обзора.

Вы увидите, что в этом случае количество многоугольников даже меньше, чем при использовании версии с удалением невидимых поверхностей, поскольку удаляются целые поддеревья, и многоугольники этих поддеревьев, имеющие видимые плоскости, не проходят через наш фильтр, как это было в более простой функции отбраковки невидимых поверхностей.

**НА ЗАМЕТКУ**

Чтобы самостоятельно скомпилировать демонстрационную программу, необходим файл DEMOII13\_3.CPP, а также модули T3DLIB1-11.CPP|H и библиотечные файлы DirectX. Кроме того, вам понадобится файл ресурсов для меню и т.п., имя этого файла — DEMOII13\_3.RC. Для экспериментов воспользуйтесь файлом DOOM01.LEV.

Еще одно заключительное замечание. Существует еще более **мощный** метод пространственной отбраковки, чем описанные. При этом рассматриваются многоугольники, а не плоскости, в которых они лежат, и используется отсечение конечной областью видимости. В результате, если мы определили, что некий многоугольник находится перед ближней или за дальней плоскостью отсечения, можно отбросить все соответствующие поддеревья, как показано на рис. 13.44.

Результаты действительно впечатляют, большего просто трудно ожидать. Возможности **BSP-деревьев** в этом смысле уникальны. Конечно, метод имеет свои ограничения, но если вы сможете использовать его в своей системе, то это будет **мощный** союзник в борьбе за скорость.

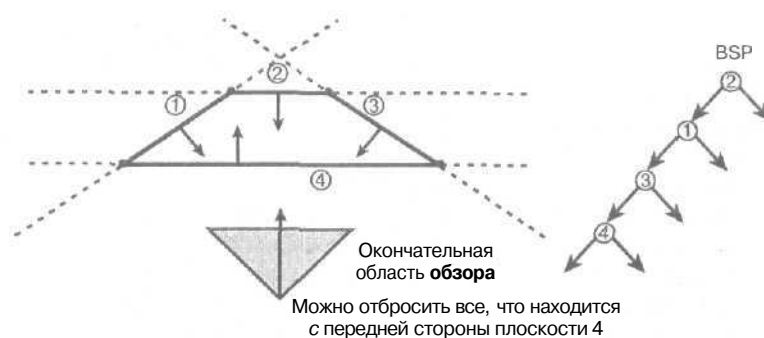


Рис. 13.44. Использование отсечения областью видимости для более эффективной отбраковки

## Использование BSP-деревьев для выявления столкновений

Наконец, рассмотрим выявление столкновений с помощью BSP-деревьев (более подробно мы остановимся на этом, когда будем рассматривать тему выявления столкновений, а не BSP-деревья). BSP-дерево можно использовать для быстрого обнаружения столкновения с игроком или другими объектами. В большинстве случаев игрок или другие объекты заключаются в **ограничивающие** объемы, такие как кубы, сферы или цилиндры, но главное, что после построения окружающей поверхности для быстрого выявления **потенциальных** столкновений используется BSP.

Преимущество BSP состоит в том, что этот метод позволяет очень быстро **совершать** обход трехмерного **пространства**, "перемещая" координаты объекта вниз по **BSP-дереву**, пока объект не окажется внутри выпуклого подпространства некоего листового узла. Такая операция требует в среднем  $\log, n$  шагов, однако в худшем случае, для полностью несбалансированного дерева может потребоваться  $n$  шагов. Результат  $\log, n$  следует из того, что на каждой итерации при сравнении точки наблюдения или центра объекта с узлами отбрасывается половина объема трехмерного пространства, и так продолжается, пока исследуемый узел не окажется листом.

После **того** как исследуемая точка окажется внутри листового узла, можно **использовать** геометрию для выяснения **того**, имеется ли (или может ли быть) столкновение объектов — путем расширения **ограничивающего** объема или расширения самого BSP-дерева вдоль векторов нормалей (т.е. каждая плоскость разделения движется вдоль ее нормали) до возможного пересечения.

## Интеграция BSP-деревьев в стандартную визуализацию

Этот раздел важен только в том случае, если вы захотите с помощью обхода BSP деревьев получать точный порядок визуализации от дальних объектов к ближним (или от ближних к дальним), как в алгоритме художника. Если вы не собираетесь пользоваться этим свойством, интеграция значения не имеет.

### Совместное использование BSP-деревьев и Z-сортировки

Совместное использование BSP-деревьев и **Z-сортировки** — достаточно сложная вещь, поскольку BSP-дерево дает точное упорядочение, а Z-сортировка может привести к нарушению существующего упорядочения многоугольников. В демонстрационной

программе редактора уровня мы уже сталкивались с подобной проблемой. Известно, что полы и потолки всегда находятся "под" геометрией стен; иными словами, как и художник, мы будем всегда сначала рисовать именно их, а уж затем визуализировать стены в порядке, предложенном BSP. Таким образом, для простых, поделенных на уровни сцен можно использовать **Z-сортировку** для фона и переднего плана, если эта сортировка не затрагивает **составляющие** многоугольники в BSP-дереве.

Например, в демонстрационной программе все многоугольники пола компланарны, поэтому не имеет значения, в каком порядке они визуализируются, так как они не могут загораживать друг друга. Однако припишем им определенную высоту — возможно, наш пол с ухабами.... В таком случае ближний многоугольник с выемкой следует рисовать только после дальнего многоугольника без выемки. Этот случай **по-прежнему** можно обработать с помощью смешанного подхода; загрузить список визуализации со всеми многоугольниками пола, выслать для них Z-сортировку, а затем включить **BSP-дерево** с его совершенным порядком. До тех пор пока в **BSP** нет ни одного объекта, меньшего объектов пола (т.е. в **BSP** нет, скажем, крошечной стены, которую могут заслонять некие неровности пола), можно по-прежнему визуализировать список от дальних объектов к ближним и получать правильный порядок.

Лично я не люблю использовать Z-сортировку совместно с **BSP-деревьями**, поскольку при этом ее практически невозможно использовать ни для чего, кроме полов, потолков или геометрических объектов, о которых достоверно известно, что они полностью находятся перед или позади объектов BSP-дерева. **Z-буферизация** является более удачным подходом.

### Совместное использование BSP, Z-буферизации и движущихся объектов

Основной целью при использовании BSP-дерева в визуализации является вычисление надлежащего порядка визуализации от дальних объектов к ближним (возможно, с использованием потенциально видимых множеств для минимизации наложения многоугольников), а затем вывод многоугольников *без* применения Z-буферизации. Однако для движущихся объектов такой подход не годится, поскольку невозможно всякий раз при перемещении объекта пересчитывать BSP-дерево (на это могут уйти секунды, минуты, а при высокой сложности уровня — даже часы).

#### НА ЗАМЕТКУ

Если многоугольник, лежащий в некоторой плоскости BSP, перемещается в той же плоскости и не попадает в другое выпуклое подпространство, пересчет BSP-дерева не требуется. Это очень важно для таких объектов как двери, лифты и т.п.

Увы, нам необходимо найти способ интеграции визуализации BSP-дерева с применением Z-буферизации. Однако при применении Z-буферизации к BSP-дереву мы потеряем весь смысл визуализации BSP. Поэтому можно воспользоваться следующим трюком: создать специальные **растеризаторы**, в которых можно отключить функции сравнения значений *z* и просто производить запись в Z-буфер. Тогда вместо того, чтобы для каждого пикселя выполнять операции *чтения*, *сравнения* и *записи*, производится только запись (*сквозная запись*). При этом Z-буфер заполняется таким образом, что его можно использовать на последующих стадиях. Правильность заполнения Z-буфера гарантируется тем, что симметричный обход BSP-дерева приводит к корректному упорядочению объектов от дальних к ближним и, следовательно, к правильному заполнению Z-буфера (рис. 13.45).

Далее представлен фрагмент функции `Draw_Triangle_2DZB_16()` перед добавлением в нее возможности сквозной записи.

```
for (xi = xstart; xi <= xend; xi++)  
{  
    // Проверка, не является ли значение z текущего пикселя
```

```

// ближе, чем текущее значение z в буфере
if (zi < z_ptr[xi])
{
    // Запись в формате 5.6.5
    screen_ptr[xi] = color;

    // Обновление Z-буфера
    z_ptr[xi] = zi;
} // if

// Интерполяция u,v,w,z
zi += dz;
} // for xi

```

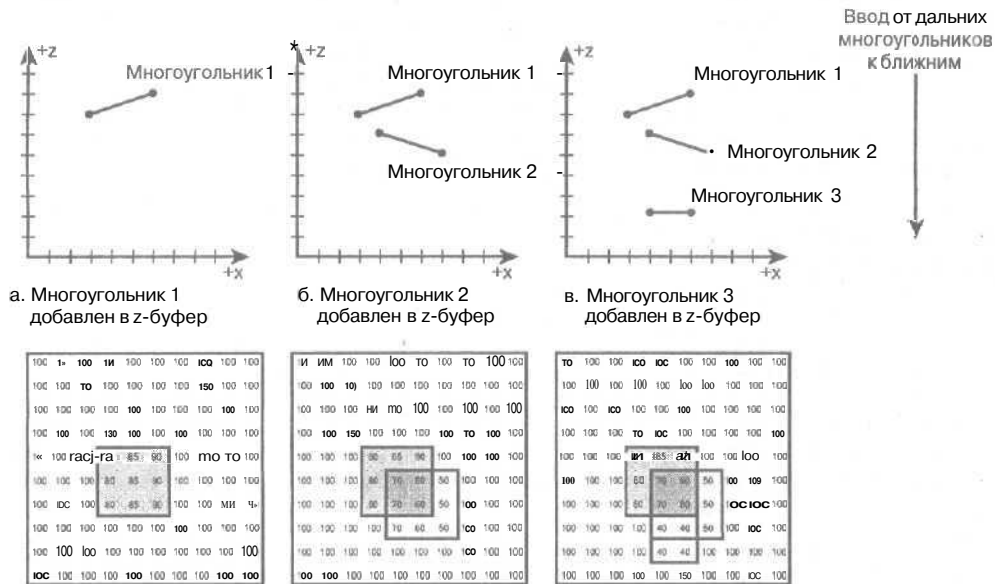


Рис. 13.45. При обходе от более удаленных к менее удаленным значения в Z-буфере всегда перезаписываются более близкими значениями z.

После удаления сравнения значений z и разрешения непосредственной сквозной записи функция примет следующий вид

```

for (xi = xstart; xi <= xend; xi++)
{
    // Сквозная запись Z-буфера
    // Запись в формате 5.6.5
    screen_ptr[xi] = color;

    // Обновление Z-буфера
    z_ptr[xi] = zi;

    // Интерполяция u,v,w,z
    zi += dz;
} // for xi

```

Теперь мы можем визуализировать **движущиеся** объекты с использованием Z-буфера. В результате мы получаем точный порядок объектов благодаря применению BSP, но при этом движущиеся объекты могут, например, скрываться за стенами.

## Код сквозной записи Z-буфера

Чтобы обеспечить возможность сквозной записи, необходимо добавить эту **функциональность** в функции растеризации (хотя бы в некоторые из них); кроме того, нам нужна новая константа контекста визуализации для выбора режима сквозной записи Z-буфера, а также новая функция контекста визуализации. Начнем с константы, позволяющей выбирать режим контекста визуализации со сквозной записью Z-буфера.

```
// Использовать Z-буфер для сквозной записи, без сравнений
#define RENDER_ATTR_WRITETHRUZBUFFER 0x00000008
```

Далее нам нужны некоторые функции растеризации с поддержкой Z-буферизации со сквозной записью. В сущности, нужно просто удалить логику проверки Z-буфера, оставив код записи в него. У нас порядка 40–50 тысяч строк кода **растеризатора**, так что я приведу новые прототипы функций только для плоского затенения, затенения по Гуро, аффинного **текстурирования** с постоянным освещением и аффинного текстурирования с плоским затенением. Имена новых прототипов отличаются от старых только добавлением букв WT (обозначающих сквозную запись — write through) в имена функций.

### Функция

```
void Draw_Gouraud_TriangleWTZB2_16(
    POLYF40V2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель видеобуфера
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель Z-буфера
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция **Draw\_Gouraud\_TriangleWTZB2\_16()** выводит затененный по Гуро треугольник с применением Z-буферизации со сквозной записью. Многоугольник не **текстурирован** и выводится только с цветом.

### Функция

```
void Draw_Triangle_2DWTZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель видеобуфера
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель Z-буфера
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция **Draw\_Gouraud\_Triangle2DWTZB\_16()** выводит треугольник с плоским затенением и с применением Z-буферизации со сквозной записью. Многоугольник не **текстурирован** и выводится одним цветом.

### Функция

```
void Draw_Textured_TriangleGSWTZB_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель видеобуфера
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель Z-буфера
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_Textured_TriangleGSWTZB_16()` выводит затененный по Гуро треугольник с аффинным отображением текстуры и Z-буферизацией со сквозной записью.

### Функция

```
void Draw_Textured_TriangleFSWTZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель видеобуфера
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель Z-буфера
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_Textured_TriangleFSWTZB_16()` выводит треугольник с плоским затенением, аффинным отображением текстуры и Z-буферизацией со сквозной записью.

### Функция

```
void Draw_Textured_TriangleWTZB2_16(
    POLYF4DV2_PTR face, // Указатель на треугольник
    UCHAR *_dest_buffer, // Указатель видеобуфера
    int mem_pitch,       // Байтов в строке, 320, 640 и т.д.
    UCHAR *_zbuffer,     // Указатель Z-буфера
    int zpitch);         // Байтов в строке Z-буфера
```

### Описание

Функция `Draw_Textured_TriangleWTZB_16()` выводит треугольник постоянным затенением, аффинным отображением текстуры и Z-буферизацией со сквозной записью.

Все эти функции работают так же, как и предыдущие версии, за исключением того, что они не проверяют значения z, а только записывают их. За корректный порядок визуализации несет ответственность вызывающая функция.

Наконец, нам нужна новая функция контекста визуализации, содержащая код для обработки нового режима Z-буферизации, `RENDER_ATTR_WRITETHRUZBUFFER`.

Вот прототип новой функции.

```
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2(
    RENDERCONTEXTV1_PTR rc);
```

Данная функция практически идентична предыдущей версии `Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16()` за исключением того, что в ней поддерживается новый режим сквозной записи в Z-буфер и известно, как вызвать соответствующие функции растеризации.

Ниже приводится пример задания контекста визуализации для обработки Z-буфера со сквозной записью.

```
// Задание контекста визуализации для 1/z буфера
rc.attr = RENDER_ATTR_WRITETHRUZBUFFER |
    RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;
```

```
// Очистить z буфер
Clear_Zbuffer(&zbuffer, (32000 << FIXP16_SHIFT));
```

```
rc.video_buffer = back_buffer;
rc.lpitch       = back_lpitch;
rc.mip_dist     = 0;
rc.zbuffer      = (UCHAR *)zbuffer.zbuffer;
rc.zpitch       = WINDOW_WIDTH*4;
rc.rend_list    = &rend_list;
```

```
rc.texture_dist - 0;
rc.alpha_override - -1;
```

```
// Визуализация сцены
Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2(&rc);
```

Единственным изменением, помимо вызова `Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2()`, является использование флага `RENDER_ATTR_WRITETHRUZBUFFER` вместо `RENDER_ATTR_ZBUFFER`. Следует, однако, отметить, что перед проходом Z-буфер по-прежнему очищается. При проходе производится сквозная запись Z-буфера, но перед записью его по-прежнему необходимо инициализировать. В противном случае следующий проход, когда объекты вставляются с включенной Z-буферизацией, будет некорректным.

Это все, что необходимо для добавления объектов. Последовательность действий будет следующей.

1. Инициализируется Z-буфер.
2. Визуализируется вся статическая геометрия в режиме сквозной записи Z-буфера. Проход должна осуществляться от более удаленных объектов к менее удаленным, поскольку Z-буфер в данном случае не будет исправлять неправильный порядок визуализации.
3. Визуализируются все движущиеся объекты с включенной Z-буферизацией.
4. Кадр выводится на экран.

### Пример с движущимися объектами

В качестве примера использования BSP при наличии движущихся объектов возьмем демонстрационный редактор уровня и создадим версию, которая загружает объект и двигает его по круговой траектории в игровом пространстве. Вектор ориентации движущегося объекта всегда направлен вдоль касательной к этой круговой траектории, как показано на рис. 13.46.

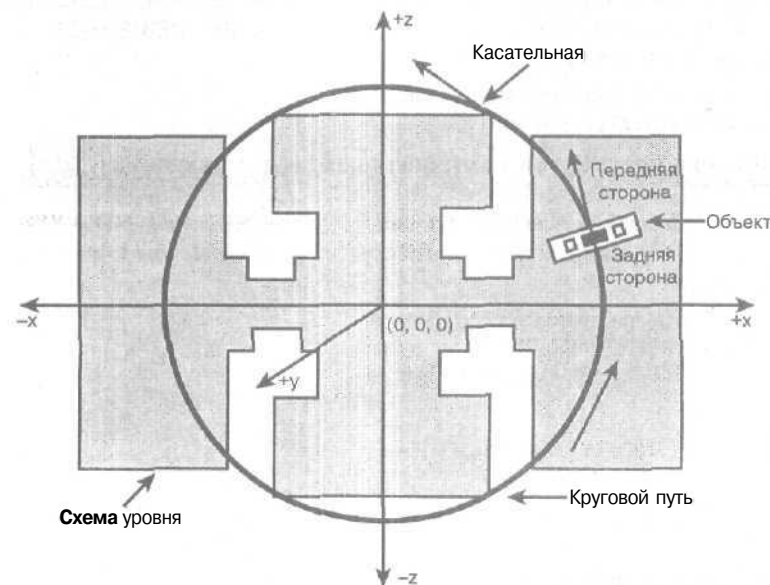


Рис. 13.46. Траектория и ориентация Z-буферизуемого объекта

Единственное изменение по сравнению с существующей версией демонстрационной программы состоит во введении нового режима сквозной записи и добавлении в код функции `Game_Main()` кода для движения объекта по кругу. Этот программный блок не является частью стандартной версии, он добавлен для того, чтобы продемонстрировать вам работу метода. Имя соответствующей программы `DEMOII13_4.CPP|EXE`, а на рис. 13.47а и 13.47б представлены копии экрана при работе программы (каркасный и сплошной режимы вывода). Программа идентична программе редактора уровня, однако при переходе в трехмерный режим с помощью команды меню **Build⇒Compile BSP and View Z-буферизуемый** объект будет двигаться по кругу. Вы можете использовать любой созданный вами уровень или, например, `DOOM01.LEV`. Чтобы запустить демонстрационную программу, нужно сначала выбрать команду меню **File⇒Load .LEV File**. Затем после загрузки файла уровня выбирается команда **Build⇒Compile BSP and View**. Попробуйте в тестовом режиме включить и отключить Z-буфер с помощью клавиши `<Z>`.

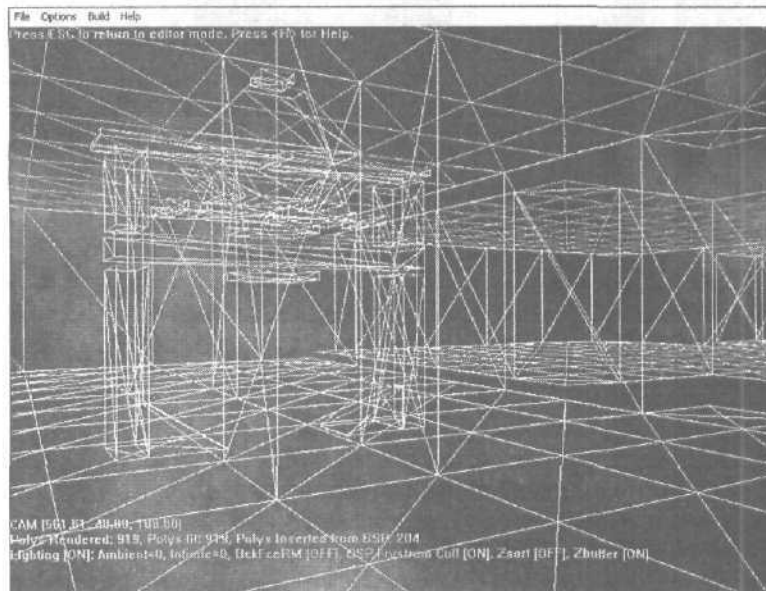


Рис. 13.47а. Копия экрана при работе демонстрационной программы с применением Z-буферизации (каркасная модель)

#### НА ЗАМЕТКУ

Чтобы самостоятельно скомпилировать демонстрационную программу, необходим файл `DEMOII13_4.CPP`, а также модули `T3DLIB1–11.CPP|H` и библиотечные файлы `DirectX`. Кроме того, вам понадобится файл ресурсов с меню `DEMOII13_4.RC`. Для проведения экспериментов необходим также файл `DOOM01.LEV`.

## Потенциально видимые множества

Как уже неоднократно подчеркивалось, возможность избежать обработки геометрических объектов, невидимых из точки наблюдения (находящихся вне области видимости) — очень ценная вещь. Мы видели, что для широкомасштабной отбраковки BSP-узлов можно использовать BSP-дерево, поскольку для любого его узла верно утверждение: если данный узел находится вне области обзора, то все, находящееся позади этого

узла, также находится вне области обзора и его можно удалять, не визуализируя. Таким способом можно удалить огромное количество геометрических объектов и ускорить процесс визуализации.

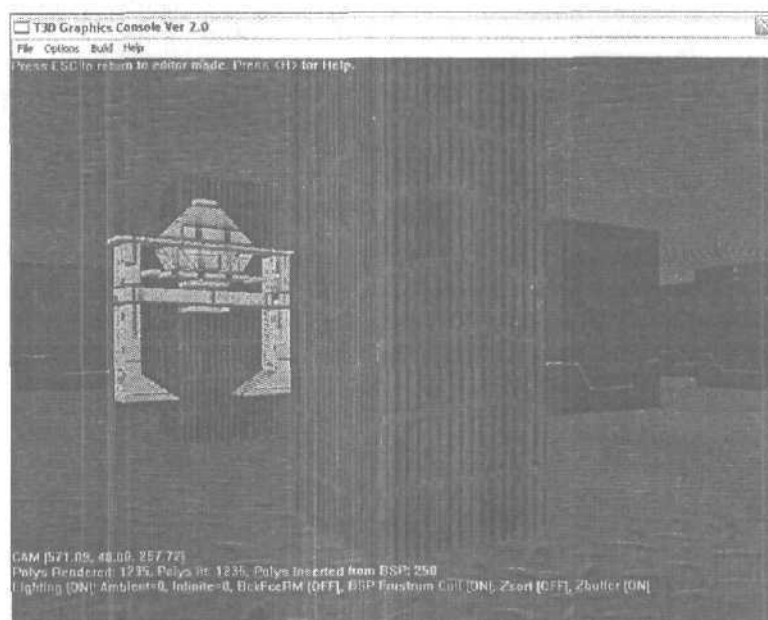


Рис. 136. Копия экрана при работе демонстрационной программы с применением Z-буферизации (сплошная модель)

Однако остается еще и потенциально видимая геометрия. Если в видимом пространстве имеется 10000 многоугольников, пусть даже с нулевым наложением, нам придется обрабатывать их и выяснять, какие из них не следует выводить. Хорошо бы иметь возможность определять заранее, какие многоугольники нужно выводить, а какие нет, т.е. для любой заданной точки наблюдения составить список всех *потенциально видимых* многоугольников. Мы можем произвести отбор среди крупномасштабных геометрических объектов, а затем, при обработке списка многоугольников, находящихся в видимом пространстве, обратиться к списку потенциально видимых множеств (potentially visible set, PVS) многоугольников, в котором точка наблюдения используется в качестве индекса, и рисовать только многоугольники из этого списка.

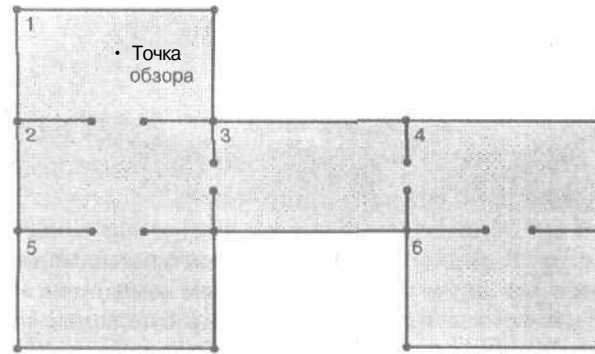
Существует множество способов реализации данной идеи — с помощью BSP, октадеревьев и другие. Остановимся подробнее на самой идее. Представим, что у нас есть некий редактор уровня и в нем создано несколько комнат, как показано на рис. 13.48.

На рисунке показаны шесть различных комнат, стены которых параллельны осям координат; каждая комната состоит из нескольких многоугольников. В комнатах могут находиться объекты, состоящие из многоугольников. Для простоты предположим, что все данные являются статическими. Создадим PVS для комнаты 1. Вопрос формулируется так: если мы находимся в комнате 1, какие комнаты являются потенциально видимыми? Как следует из рисунка, из комнаты 1 через дверные проемы можно увидеть комнаты 1, 2, 3 и 5.

НА ЗАМЕТКУ

Комната 1 включена в этот перечень, поскольку многоугольники комнаты, в которой находится точка наблюдения, всегда видимы.

а. Видимость комнат из комнаты 1



б. Структура данных для информации о видимости

Исходная комната	{	Комната 1: 1, 2, 3, 5	Список видимых комнат для каждой комнаты
		Комната 2: 2, 1, 3, 4, 5	
		Комната 3: 3, 1, 2, 4, 5, 6	
		Комната 4: 4, 2, 3, 6	
		Комната 5: 5, 1, 2, 3	
		Комната 6: 6, 4, 3	

Рис. 13.48. Видимость из комнаты в комнату

Теперь представим, что у нас есть некая иерархия, в которой каждая из комнат хранится в структуре данных совместно со списком всех потенциально видимых из нее комнат, как показано на рис. 13.48б. В этом случае получается список или массив комнат, видимых из любой комнаты-точки наблюдения. Другой способ состоит в создании матрицы смежности  $m \times m$ , где  $m$  — число комнат, а значение элемента  $ij$  ( $i$  — столбец, а  $j$  — строка) показывает, видна ли из комнаты  $i$  комната  $j$ . Запись нашего примера (рис. 13.49) с помощью матрицы смежности выглядит следующим образом.

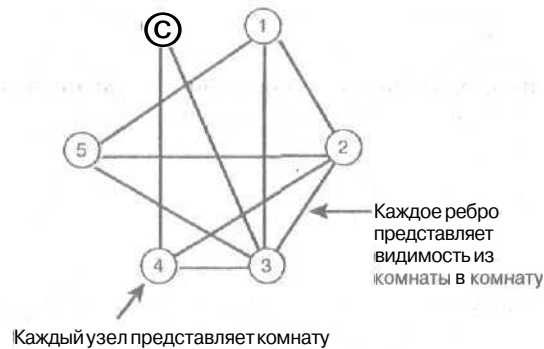


Рис. 13.49. Граф смежности комнат

	$i_1$	$i_2$	$i_3$	$i_4$	$i_5$	$i_6$
$j_1$	1	1	1	0	1	0
$j_2$	1	1	1	1	1	0
$j_3$	1	1	1	1	1	1
$j_4$	0	1	1	1	0	1
$j_5$	1	1	1	0	1	0
$j_6$	0	0	1	1	0	1

Как правило, из любой комнаты большинство других комнат данного уровня являются потенциально видимыми. В данном случае меньше всего потенциально видимых комнат для комнаты 6. Таким образом, у нас есть PVS на уровне комнат или, иными словами, мы должны добавить все многоугольники исходной комнаты к потенциально видимым комнатам. Мы еще вернемся к этому немного позже, а сейчас обсудим использование PVS.

## Использование потенциально видимого множества

Предположим, что мы создали уровень, содержащий множество **комнат**; для каждой комнаты известно, какие **другие** комнаты потенциально видимы из нее. Тогда PVS можно использовать для того, чтобы уменьшить наложение при визуализации. Предположим, например, что для представления каждой отдельной комнаты использован метод BSP или другой метод разбиения пространства. Все, что можно удалить из области обзора, уже удалено, в том числе использована отбраковка с **помощью BSP-деревьев** для удаления значительных частей среды. После этого можно отправлять **многоугольники** в конвейер для отсеивания, освещения и т.д. И **теперь** наступает очередь PVS. Сначала следует определить, в какой комнате находится камера. Это можно сделать с помощью BSP или просто разбив игровой уровень на сектора и определив, в какой ячейке находится наблюдатель. Пусть это будет комната 6, а область обзора выглядит так, как показано на рис. 13.50.

Даже после отбраковки с использованием BSP в список визуализации вероятно, попадет вся база данных моделей. Однако использование PVS позволит нам избавиться от лишнего **вывода**. Для этого достаточно найти текущую комнату, в которой находится точка наблюдения (в данном случае это комната 6). Затем производится обращение к структуре данных, содержащей PVS для данной комнаты (в данном случае мы используем список **смежности**, находим столбец 6 и считываем все потенциально видимые комнаты; 3, 4 и 6 (исходная комната всегда включается в список **смежности**)).

Поразительно, что с точки зрения **пространства** обзора, каждая отдельная комната является видимой, поэтому в конвейер попадут практически все многоугольники. Некоторые наверняка будут усечены, но многие многоугольники, полностью закрытые стенами комнаты 6, будут рисоваться или, по крайней мере, подвергаться проверке на глубину.

Это совершенно неприемлемо, и если игровой уровень напоминает показанный на рис. 13.51 уровень со множеством комнат (достаточно простой уровень *Doom*), то наложение при визуализации накапливается очень быстро и экран перезаписывается множество раз. Использование PVS в такой ситуации является незаменимым средством ускорения визуализации и сведения наложения практически к нулю.

Некоторые частично заслоненные препятствиями многоугольники по-прежнему будут включены в PVS. Это вполне допустимо, и, как следует из моего опыта, в худшем случае при использовании описанного нами PVS общего вида наложение составляет 50-150%. Без PVS на больших уровнях количество обрабатываемых многоугольников легко может увеличиться в 10-20 раз. Как видите, **использование** метода PVS в ряде ситуаций может на порядок увеличить скорость кадров (или, что то же самое, уменьшить количество обрабатываемых многоугольников).

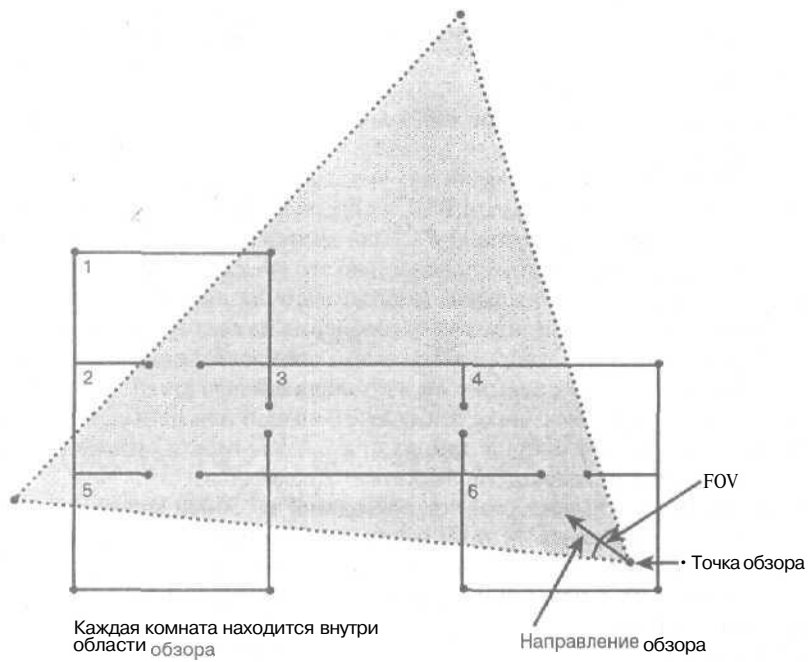


Рис. 13.50. Использование PVS для отбраковки геометрических объектов

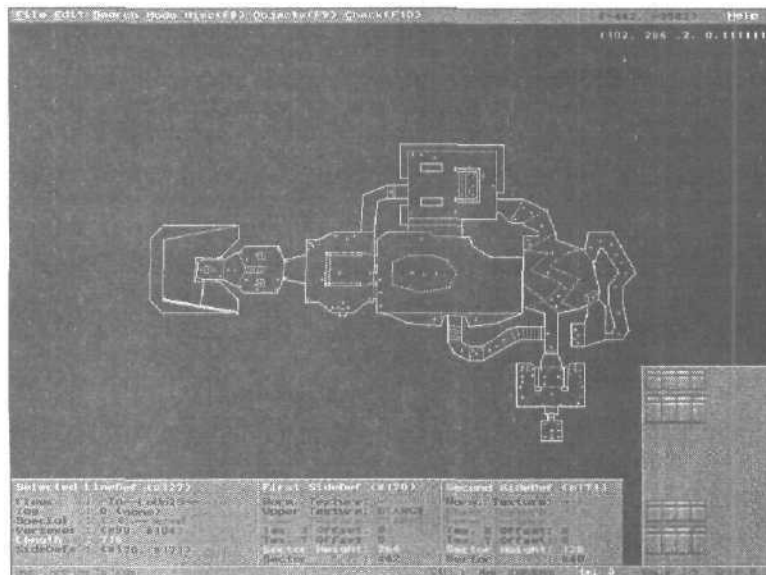


Рис. 13.51. Даже в простом уровне игры типа Doom может возникнуть огромное количество наложений

## Методы представления потенциально видимого множества

При больших размерах уровней (от 10000 до 1000000 многоугольников) размеры PVS также могут значительно увеличиться, и хранение и доступ к множеству осложнится. В нашем простом примере мы использовали два метода: **явно** заданный список, связанный с каждой комнатой, сектором или узлом BSP, и список смежности. Любой из этих методов будет хорошо работать при кодировании PVS для комнат (т.е. PVS, описывающего видимость комнат, а не отдельных многоугольников), но это уже **следующий** уровень PVS.

Если же мы хотим описать отдельные многоугольники, видимые из любой точки обзора, можно вновь воспользоваться методом разбиения на сектора, когда точка наблюдения указывает определенный сектор, а затем вычисляются все **многоугольники**, видимые из данной точки. Нахождение видимых многоугольников является геометрической задачей, к которой мы **еще** вернемся, но ее решение в большей или меньшей степени основано на видимости источников света и линиях прямой видимости, что не так уж сложно. Остановимся на кодировании такого большого объема данных.

Предположим, что у нас есть **уровень**, состоящий из 50000 многоугольников и 100 комнат. У нас есть два варианта. Можно кодировать множество потенциально видимых объектов на основе комнат, исходя из того, в какой комнате находится точка наблюдения, или вычислять ее на основании разбиения игрового пространства на сектора, представляющие собой небольшие квадраты или кубы, а затем для каждой комнаты или сектора составлять список **потенциально** видимых многоугольников во всем **пространстве**. Эта схема показана на рис. 13.52.

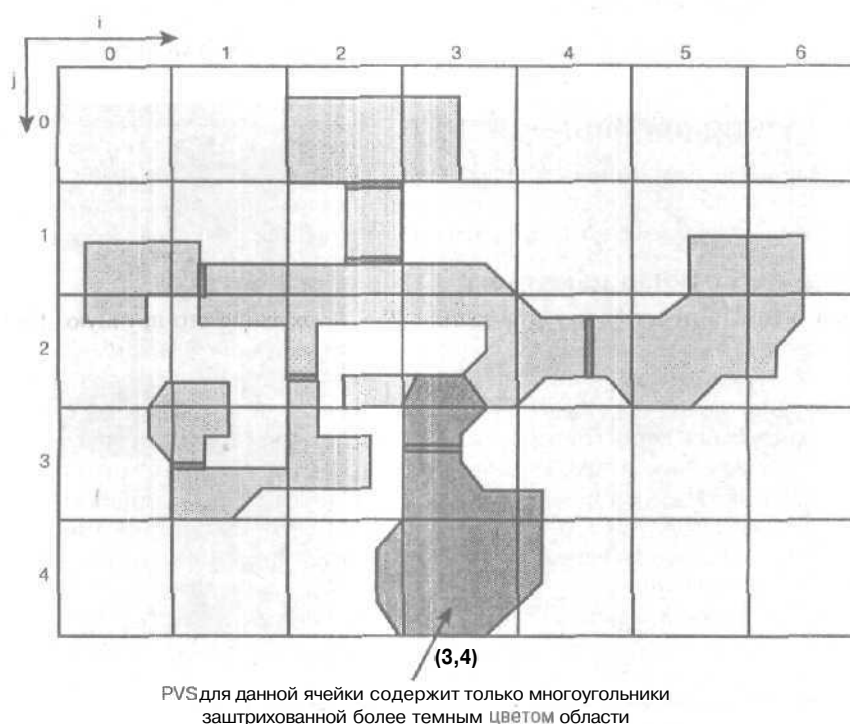


Рис. 13.52. PVS на основе секторов

Здесь налицо очевидный компромисс. Если кодировать PVS как указатели или индексы, то каждому многоугольнику в PVS для данной комнаты или сектора должен соответствовать свой индекс. Так, в нашем примере со 100 комнатами насчитывается 50000 многоугольников, в среднем 1000 из них видимы из каждой комнаты. Это означает, что PVS будет иметь размер **следующего** порядка;  $(100 \text{ комнат}) \times (1000 \text{ потенциально видимых многоугольников}) \times (4 \text{ байта}) = 400 \text{ Kb}$  для всего PVS.

В действительности это не так уж плохо. Естественно, можно немного улучшить результат, если использовать для каждого индекса три байта вместо четырех, но более эффективно кодировать PVS другим способом.

Например, вместо использования явного списка индексов можно применить следующий метод кодирования: в качестве элемента PVS создается упорядоченный список битов  $0\dots p$ , соответствующих многоугольникам; бит устанавливается, если соответствующий многоугольник является видимым. Например, если уровень состоит из 10 многоугольников и для комнаты  $p$  видимыми являются многоугольники 0, 5, и 6, PVS будет выглядеть **следующим** образом:

PVSкомнаты $p$ : 1, 0, 0, 0, 0, 1, 1, 0, 0, 0

При таком способе кодирования нельзя пропускать индексы, поскольку они упорядочены, но память при этом все равно экономится. Посмотрим, что даст такой способ кодирования в нашем случае:

$(100 \text{ комнат}) \times (50000 \text{ многоугольников}) \times (1 \text{ бит}) = 625000 \text{ байт!}$

Где же обещанная экономия? В упаковке! Простое RLE-кодирование, объединяющее нули или единицы, позволяет в большинстве случаев сжать записанное в таком виде множество с коэффициентом сжатия от 1 до 20, т.е. реально потребуется порядка 50Kбайт памяти.

Выбор, в каком виде хранить PVS (в виде индексов многоугольников или явного перечня доступных многоугольников в упорядоченном списке), остается за вами. Оба метода имеют свои за и против. Лично я предпочитаю список индексов,

## Более точное вычисление PVS

До сих пор мы не рассматривали подробности вычисления PVS, но не потому, что это сложно, а потому, что не **существует** единственно правильного способа сделать это. Существует целый ряд подходов; остановимся на двух наиболее популярных среди них.

### Создание PVS с помощью программных средств

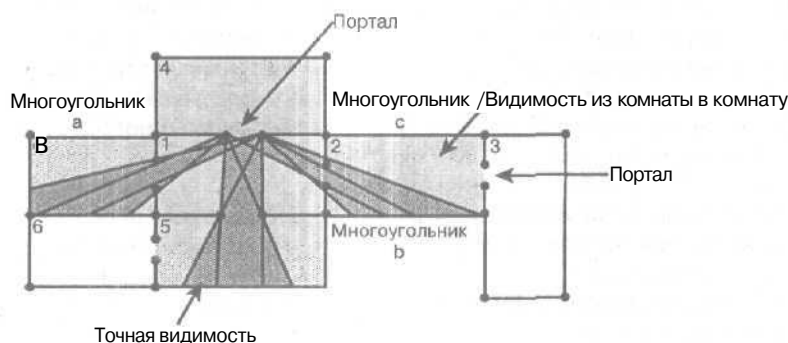
Первый и самый простой метод создания PVS — создание его вручную. Предположим, например, что вы создали некую программу для рисования двумерных миров, которые затем преобразуются в трехмерные. Каждая комната состоит из множества многоугольников, и после моделирования двумерного мира происходит переход к следующей стадии моделирования, когда вручную создается список смежности или другая структура, **содержащая** информацию о том, какие комнаты видны из других. Этот список можно вычислить с помощью отслеживания линий обзора, методом заливки или **каким-нибудь** иным способом. В чем же тут сложность? При работе с небольшими мирами, состоящими из 10–50 комнат, можно легко и быстро составить список смежности, просто визуально просматривая комнату за комнатой и фиксируя соответствующую информацию без всяких **сложных** алгоритмов.

Именно этот метод я предлагаю использовать вначале: просто закодируйте PVS вручную или с помощью некоторого инструментального средства, чтобы вы могли сосредоточиться на использовании PVS, а не его создании. Поскольку PVS обычно создается до игры, время работы над ним не играет ключевой роли. Но если вы постоянно меняете иг-

ровый мир, то перестройка PVS может оказаться непростым вопросом. Метод PVS служит исключительно для оптимизации, так что лучше подождать до тех пор, пока вы не будете полностью удовлетворены созданным миром игры, и лишь затем создавать PVS.

### Создание PVS с использованием источников света и порталов

Следующий, более широко применяемый метод генерации PVS состоит в использовании свойств видимости через порталы или в способности света "находить" потенциально видимые из каждой комнаты (сектора, ячейки) многоугольники. На рис. 13.53 изображены комнаты и помечены все двери, которые называются *порталами*.



Многоугольники a, b, c включены в PVS на основе комнат,  
но не в PVS на основе многоугольников

Рис. 13.53. Видимость через порталы с разрешением на уровне многоугольников

Каждая комната построена из отдельных многоугольников, поэтому мы будем строить PVS, состоящее из многоугольников. На рисунке менее темным цветом показаны видимые области на уровне комнат из комнаты 4. Поскольку нас интересует более точная информация о видимости, нам следует работать с отдельными многоугольниками. Для этого для каждого светового луча, не встречающего никаких препятствий на пути из комнаты 4 в другие заштрихованные комнаты, вычисляется его путь (сделать это вручную практически невозможно, но если представить, что портал из комнаты 4 является источником **света**, а затем рассмотреть прохождение световых лучей через другие **порталы**, то получится именно то, что мы ищем (как показано более темным цветом на рис. 13.53)).

На рисунке из портала комнаты 4 через смежную комнату 1 в другие **потенциально** видимые комнаты проведены пучки лучей, и пространство, куда попадает свет, и определяет объем, потенциально видимый из комнаты 4. Любая грань, оказавшаяся внутри этих заштрихованных темным цветом областей, является частью PVS для комнаты 4. Кроме того, к данному списку необходимо полностью добавить комнату 1, поскольку она непосредственно соседствует с комнатой 4 и между ними нет промежуточных комнат с порталными ограничениями.

С помощью этого нового метода нам удалось минимизировать PVS комнаты 4 — от содержащего все геометрические объекты комнат 0, 1, 2 и 5, до множества, которое содержит только объекты, находящиеся внутри конусов света, образуемых при использовании портала комнаты 4 в качестве источника. В результате в данном случае для комнаты 4 в PVS будет включено на 30-50% меньше многоугольников. Например, многоугольники a, b и c теперь не входят в PVS, несмотря на то, что они входили во множество PVS на уровне комнат, построенное при помощи более грубого метода.

Этот метод распространяется и на объекты, *находящиеся* в комнате. Эти объекты также являются многоугольниками и для них также можно проверить, находятся ли они внутри световых конусов. Естественно, что PVS хороши только для статических объектов, но если объекты движутся по комнате, то они могут становиться невидимыми, хотя PVS будет по-прежнему содержать их.

Главное — это то, что вычисление PVS происходит вне игры, так что его созданием можно заниматься столько, сколько нужно, чтобы определить многоугольники, видимые из любой заданной точки или комнаты.

Эти рассуждения натолкнули меня на мысль — что делать, если у вас нет разбитого на комнаты пространства с четко определенными порталами, которые можно использовать в качестве источников света? Ничего страшного, нужно просто проявить сообразительность. Один возможный метод состоит в разбиении пространства на **квадраты** или кубы (в зависимости от того, является ли оно в основном поделенным на **уровни** или произвольным трехмерным). Сначала случайным образом (или по определенному **правилу** — в зависимости от того, как может двигаться игрок) выбираются тысячи, десятки тысяч или сотни тысяч точек обзора в игровом пространстве. Затем из каждой точки наблюдения необходимо попытаться построить лучи к каждому многоугольнику игрового пространства. Если удастся без помех провести хотя бы один луч от данной точки наблюдения хотя бы к одной вершине, можно не сомневаться, что данный многоугольник видим из указанной точки наблюдения и его можно добавить к PVS для данной точки. Этот процесс показан на рис. 13.54.

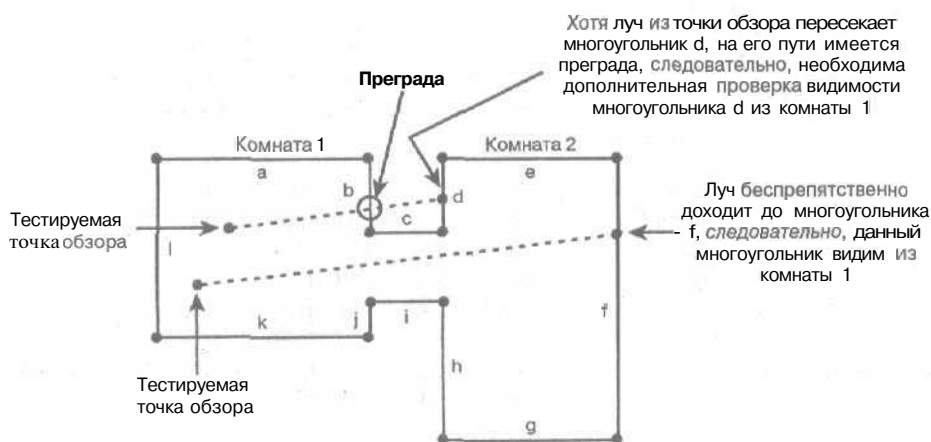


Рис. 13.54. Проведение лучей для определения видимости

## Заключительные замечания о PVS

Потенциально видимые множества абсолютно необходимы в играх с базой данных, *состоящей* из большого количества многоугольников. Их можно использовать в ходе игры практически без ограничений, и они занимают средний объем памяти даже при самом грубом методе кодирования. PVS используются совместно с BSP-деревьями, октадеревьями и (особенно) порталами. Вычисление PVS может потребовать значительного объема вычислений, в зависимости от того, какой точности вы хотите *добиться* от данного метода. Наконец, после вычисления PVS его можно закодировать как матрицу смежности, список или при *помощи* иного способа битового кодирования.

## Порталы

В последнее время порталам уделяется много внимания. Две наиболее известные игры с порталами — *Descent* и *Duke Nukem 3D*, показанные на рис. 13.55 и 13.56, соответственно. *Descent* — это трехмерная игра с шестью возможными направлениями полета, в которой игроку разрешается летать по туннелям практически неограниченного трехмерного лабиринта. В игре используется подсветка, отображение текстур и множество действительно красивых эффектов. Но самым поразительным является то, насколько быстро визуализируются игровые миры. *Duke Nukem 3D* — еще одна потрясающая трехмерная игра середины 90-х. В ней также имеется освещение в реальном времени, наложение текстур; в игре множество уровней, лестниц, движущихся объектов и т.д. — и все это продукт технологии порталов. В порталах нет ничего магического, это просто результат реализации стратегии PVS в реальном времени.

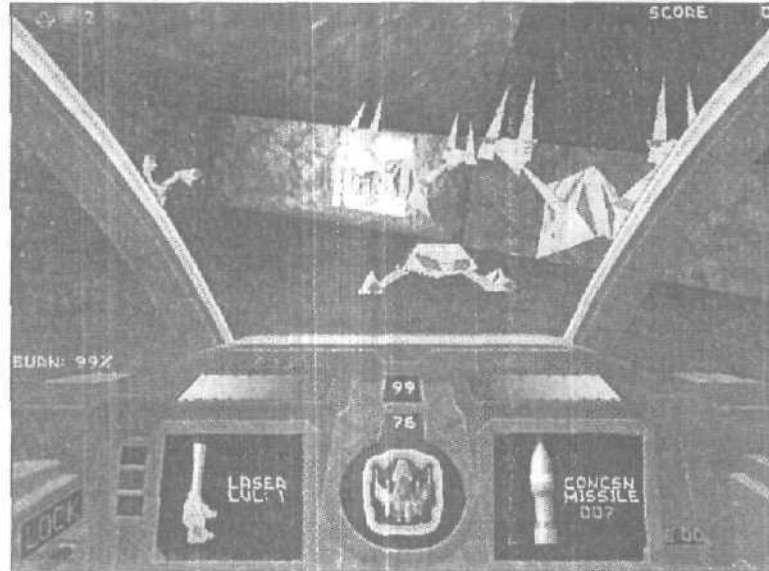


Рис. 13.55. В игре *Descent* широко используются порталы

Основная идея заключается в том, что даже если уровень состоит из миллиарда многоугольников, а в каждом кадре нужно изображать только пару сотен, то визуализировать надо только те объекты, которые можно видеть. Технология порталов основана на том, что если у вас имеется поделенная на ячейки или комнаты игровая среда и вы можете отметить "порталы" (переходы) в смежные комнаты, то можно использовать один хитрый алгоритм для вычисления PVS "на лету". Покажем, как он работает. На рис. 13.57 представлен еще один пример множества комнат. Теперь комнат всего пять и все дверные проемы отмечены как порталы.

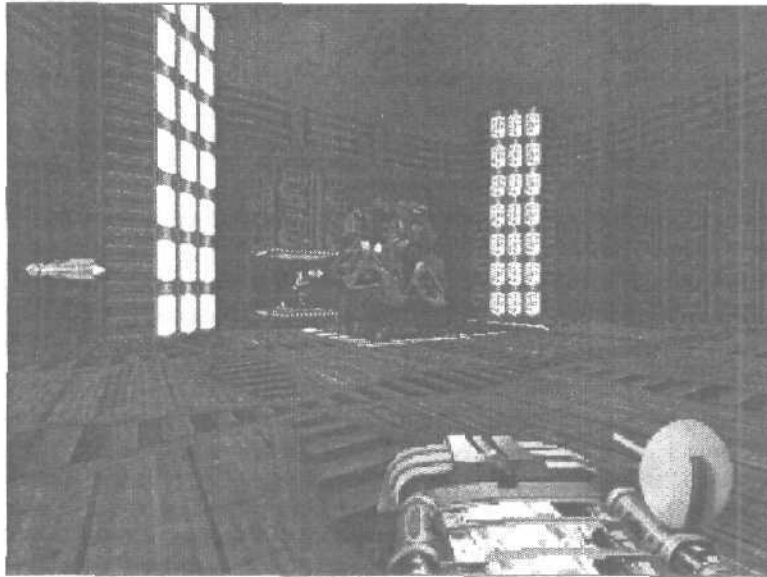
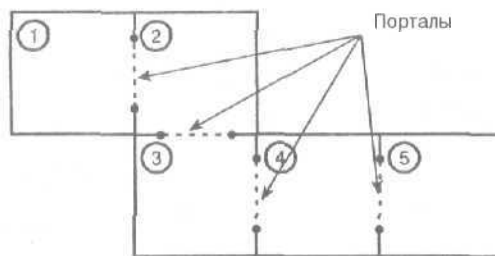


Рис. 13.56. Игра Duke Nukem 3D

а. Геометрия комнат



б. Списки смежности



Примечание: исходная комната не включается  
в список смежности

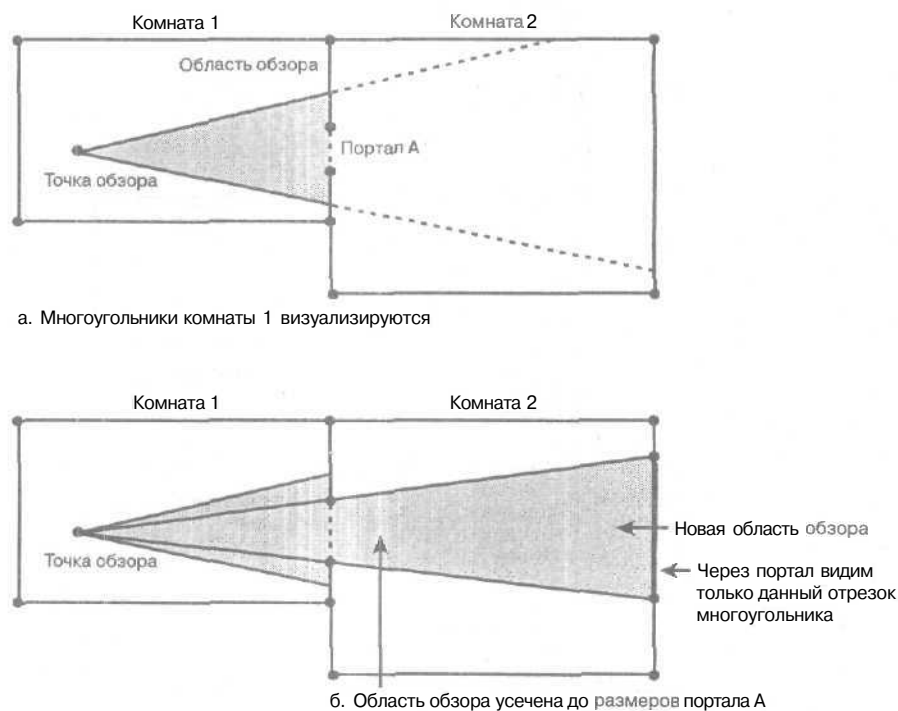
Рис. 13.57. Предвычисление видимости между комнатами при наличии порталов

Независимо от того, какие объекты могут находиться внутри комнат, давайте попытаемся вычислить видимые многоугольники "на лету", используя порталы дверных проемов. Первый шаг состоит в создании списка видимости из комнаты в комнату до начала

игры. Это следует сделать, потому что нет смысла прибегать к логике порталов, если комната 5 не видна из комнаты 1. В этом случае лучше использовать список смежности (рис. 13.57б), а не матрицу. После создания такого списка можно переходить в режим реального времени. В реальном времени выполняются следующие действия.

1. Для текущей точки наблюдения следует вычислить, в какой комнате/ячейке она находится, и сохранить данную информацию.
2. Визуализировать многоугольники **текущей** комнаты, используя любой метод (BSP, Z-буферизацию, Z-сортировку и т.д.).
3. **ВАЖНО!** Для каждого портала, ведущего из текущей комнаты (вспомните — у нас есть их список), следует откорректировать область видимости путем добавления плоскостей отсечения, которые определяют дверной проем портала, а затем рекурсивно перейти в следующую комнату списка и вывести многоугольники **из** этой комнаты.

Я подчеркнул наиболее важную часть методики, однако прежде чем углубляться в детали процесса отсечения, рассмотрим весь процесс. Начнем с комнаты, в которой находится точка наблюдения, визуализируем ее, а затем выполним рекурсивный обход комнат из списка смежности для данной комнаты, пользуясь списком порталов. Каждый раз при переходе в следующую комнату из предыдущей визуализируется только то, что видимо при наблюдении через портал из комнаты-точки наблюдения в комнате-предмете наблюдения. Иными словами, происходит ограничение поля зрения из комнаты-точки наблюдения в комнату-предмет наблюдения выходным порталом, как показано на рис. 13.58.



**Рис. 13.58. Простой пример портального отсечения и обхода**

Чтобы изучить данный процесс, рассмотрим пример всего с двумя комнатами в двумерном пространстве. Начнем со списка смежности из комнаты 1 в комнату 2 (рис. 13.58), каж-

дая комната видна из другой комнаты через дверной проем (портал). Точка наблюдения находится в комнате 1, поэтому визуализируем все многоугольники комнаты 1 с использованием полной области обзора. Переходим в комнату 2, поскольку она видима через портал, обозначенный на рисунке буквой А. Однако при этом мы ограничиваем область обзора плоскостями отсечения, определяемыми исходной точкой и геометрией портала (в данном случае двумерной). Новая область видимости показана на рис. 13.6. Из исходной точки наблюдения видимы только многоугольники этой новой области видимости, все остальные многоугольники невидимы, поэтому их не нужно визуализировать.

Здесь сразу же очевидно наличие двух проблем. Во-первых, технология порталов не строит PVS, с ее помощью всего лишь создается новый объем отсечения или область видимости. Однако если у нас есть BSP или октадерево, для текущего портала мы можем отбросить целые комнаты или ячейки, чтобы сэкономить время. Вторая проблема более серьезна. При рекурсивном переходе в следующие комнаты и добавлении к текущему видимому пространству геометрии портала область видимости постепенно превращается из симпатичной четырехугольной пирамиды в  $n$ -гранный объем, как показано на рис. 13.59.

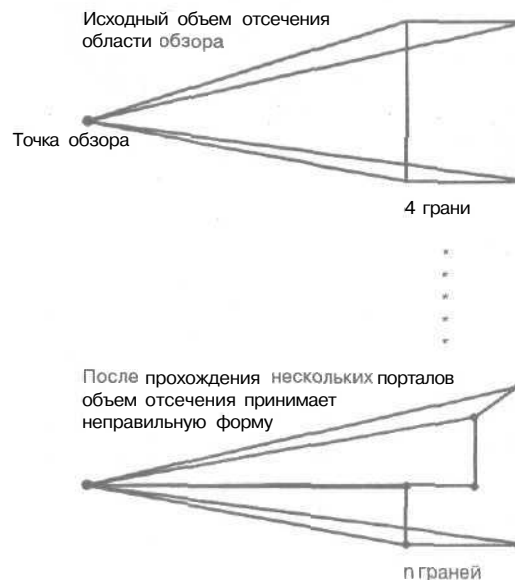


Рис. 13.59. Усложнение формы области видимости

Проблема состоит в выполнении корректного отсечения. Оно занимает так много времени, что с тем же успехом можно было просто визуализировать лишние многоугольники...

В заключение отметим, что порталы хорошо работают в простых мирах, соединенных коридорами или имеющими однообразную геометрию. В целом же порталам не удастся превзойти PVS, поскольку всегда лучше *заранее знать*, что надо будет выводить, а что — нет.

## Ограничивающие иерархические объемы и октадеревья

Ограничивающие иерархические объемы (bounding hierarchical volumes, BHV) — это еще один метод группировки объектов для быстрого обнаружения столкновений или отсечения. Фактически мы уже использовали ограничивающие сферы с самого начала,

чтобы объединять объекты для быстрой отбраковки. В методе BVH эта идея поднимается на новый уровень и создается иерархия объемов, как показано на рис. 13.60. Объемы могут представлять собой **ограничивающие сферы** (как на рисунке), иметь форму кубов, грани которых параллельны осям, или форму, ориентированную с учетом ориентации объекта. Выбор остается за разработчиком. Чаще всего используются сферы и кубы, грани которых параллельны осям.

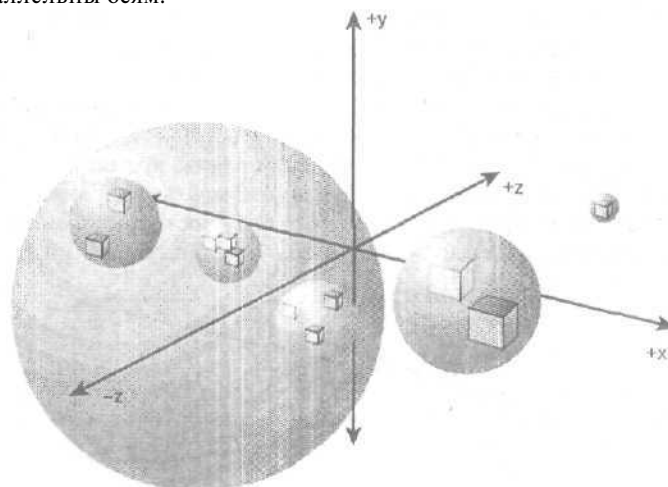


Рис. 13.60. Группировка объектов в ограничивающих иерархических сферах

В качестве примера представим, что у нас имеется набор объектов, представленный на рис. 13.61. Каждый объект представлен своей ограничивающей сферой. Идея метода BVH состоит в дальнейшей группировке объектов в большие множества с тем, чтобы можно было легко отбрасывать целое множество, а не отдельные его **составляющие**. На рис. 13.61б показаны объекты, объединенные в несколько сфер меньшего размера, а на рис. 13.61в объекты заключены в еще меньшие сферы.

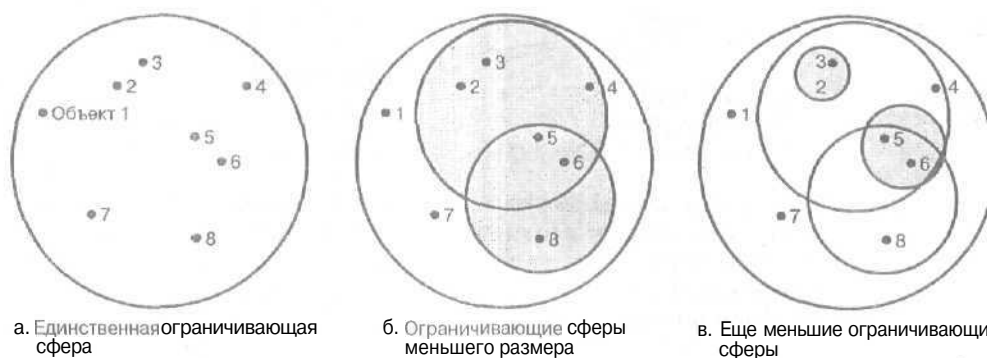


Рис. 13.61. Ограничивающие объемы различного уровня

Структура данных для хранения этих объемов может быть любой, но наиболее естественно выглядит древовидная структура. Однако поскольку каждый более высокий уровень иерархии может иметь более двух дочерних элементов, необходимо использовать **п-древовидную** структуру, как показано на рис. 13.62.

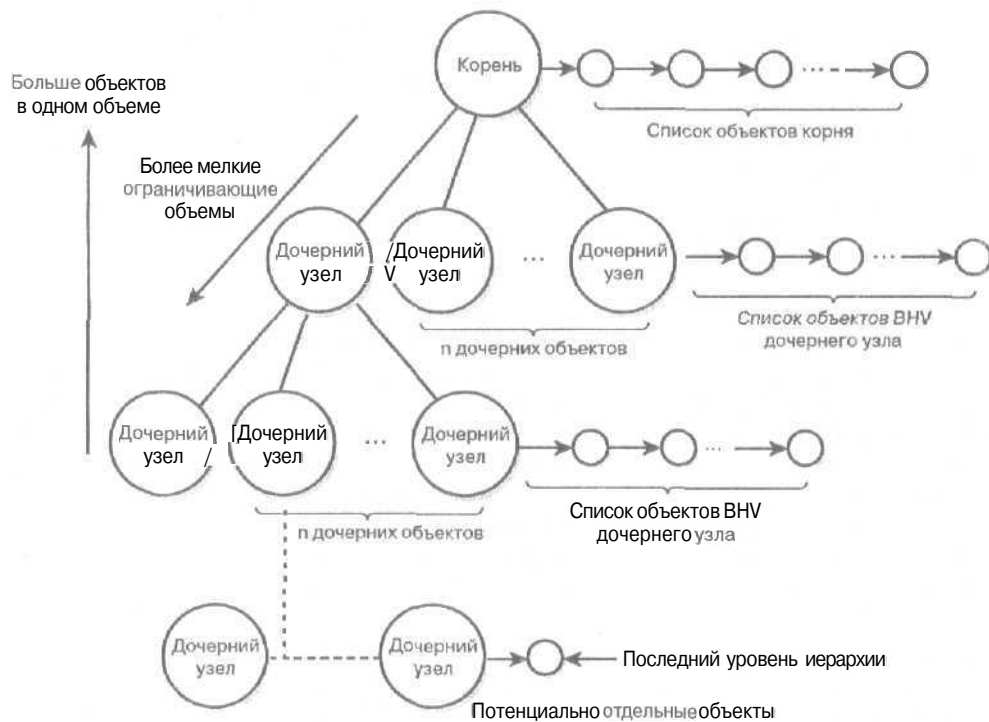


Рис. 13.62. Возможная абстрактная структура данных для хранения BVH

Каждый узел должен содержать информацию о центре, радиусе, массив или связанный список своих дочерних узлов и массив или связанный список объектов, содержащихся в данном узле вместе с информацией об их состоянии. Вот один из возможных наборов структур данных для хранения BVH.

```
// Объект-контейнер для хранения любого объекта
typedef struct OBJ_CONTAINERV1_TYP
{
    int state;           // Состояние объекта
    int attr;            // Атрибуты объекта
    POINT4D pos;         // Положение объекта
    VECTOR4D vel;        // Скорость объекта
    VECTOR4D rot;        // Вращательная скорость объекта
    int auxi[8];         // Вспомогательный массив из 8 целых
    int auxf[8];         // Вспомогательный массив из 8 чисел с
                        // плавающей запятой
    void *aux_ptr;       // Вспомогательный указатель
    void *obj;           // Указатель на главный объект
} OBJ_CONTAINERV1, *OBJ_CONTAINERV1_PTR;

// Узел BVH
typedef struct BVH_NODEV1_TYP
{
    int state;           // Состояние узла
```

```

int attr; // Атрибуты узла
POINT4D pos; // Центр узла
VECTOR4D radius; // x,y,z радиус узла
int num_objects; // Число объектов, содержащихся в узле
OBJ_CONTAINERV1 *objects[MAX_OBJECTS_PER_BHV_NODE];
// Объекты
int num_children; // Число дочерних узлов
BHV_NODEV1_TYP *links[MAX_BHV_PER_NODE];
// Ссылки на дочерние узлы

} BHV_NODEV1, *BHV_NODEV1_PTR;

```

Представленные структуры могут работать со сферами, кубами или прямоугольными формами. Поскольку радиус представлен типом `VECTOR4D`, мы можем легко получать различную протяженность вдоль разных осей. Заметим также, что `BHV_NODEV1` содержит обобщенный объект `OBJ_CONTAINERV1`, а не `OBJECT4DV2`. Кроме того, `OBJ_CONTAINERV1` содержит указатель `void*` на указываемый объект. Таким образом, при изменении объекта можно продолжать использовать те же структуры данных.

Естественно, нам нужна стратегия построения ограничивающих сфер, иными словами, каким образом следует выбирать объекты на каждой итерации и как вычислять размеры сфер. Однако сначала обсудим использование `BHV`-дерева.

## Использование `BHV`-дерева

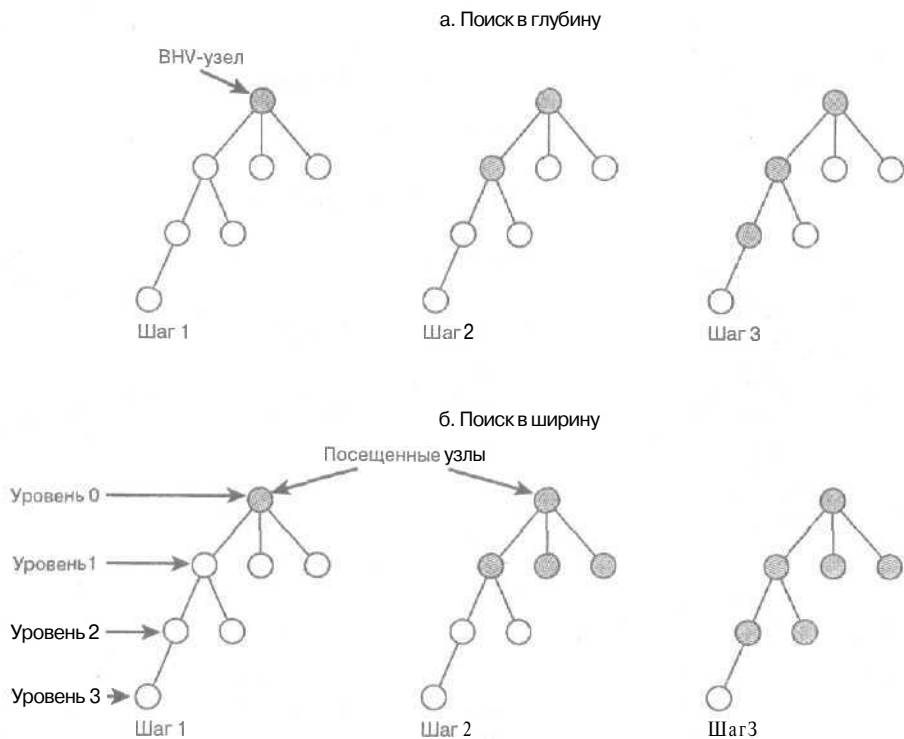
Предположим, что у нас есть `BHV`-дерево, построенное с использованием ограничивающих сфер (или кубов). Такое дерево можно использовать в процессе работы игры для осуществления крупномасштабного определения столкновений или отбраковки. Поскольку эти понятия связаны и тема данной главы — визуализация, а не столкновения, остановимся на свойствах `BHV` применительно к отбору.

`BHV` представляет всю геометрию игрового мира на различных уровнях грануляции. На самом верхнем, корневом уровне имеется единственная сфера, включающая в себя все пространство, поэтому первый шаг состоит в том, чтобы попытаться отбросить всю сферу из области видимости. Это осуществляется просто: процедура аналогична удалению ограничивающих сфер объектов. Если корневой узел может быть полностью отброшен, все объекты массива `objects[]` помечаются как отброшенные и обработка на этом заканчивается. Следующая единственная строка кода устанавливает соответствующий флаг для любого объекта.

```
SET_BIT(obj->state, OBJECT4DV2_STATE_CULLED);
```

Если самый верхний узел нельзя отбросить, реализуется явный или рекурсивный алгоритм, который обрабатывает все ссылки текущего узла и пытается отбросить ограничивающие сферы каждого узла. В каждом узле процесс продолжается в глубину или в ширину, как показано на рис. 13.63. В любом случае, если необходимо, проверяется все дерево. Следует помнить, что после отбрасывания узла все объекты данного узла помечаются как отброшенные и все дочерние узлы данного узла не посещаются.

Когда процесс завершается, все отброшенные объекты будут помечены, поэтому в ходе визуализации в основном цикле, чтобы узнать, был ли объект уже отброшен, достаточно проверить флаг `OBJECT4DV2_STATE_CULLED` в состоянии объекта. Если объект не был отброшен, необходимо выполнить стандартную процедуру отбраковки, поскольку нет гарантии, что в `BHV`-системе созданы ограничивающие объемы вокруг каждого отдельного объекта.



**Рис. 13.63. Поиск BVH в глубину и в ширину**

## Производительность

Давайте немного поговорим о вопросах производительности и чем здесь могут помочь BVH. Рассмотрим пример, в котором равномерно распределены в пространстве 10000 объектов. Теперь представим, что у нас есть четырехуровневые ограничивающие объемы. Уровень 0 является корневым и содержит все объекты. Уровень 1 также содержит все объекты, но размеры ограничивающих объемов составляют не более 50% размеров объемов корневого уровня. Та же закономерность прослеживается и для последующих уровней.

**Корень.** BVH имеет радиус  $r$ , содержит все 10000 объектов.

**Уровень 1.** Этот уровень содержит BVH с радиусом  $< r/2$ . Может иметься один или несколько BVH уровня 1. Предположим, что в данном случае на уровне 1 есть пять BVH с радиусом  $r/2$  каждый. Объекты распределены равномерно, так что в каждом BVH содержится 2000 объектов.

**Уровень 2.** На этом уровне содержатся BVH с радиусом  $< r/4$ . На этом уровне также может быть как один, так и несколько BVH. Пусть в нашем случае уровень 2 содержит 20 BVH, с радиусом  $r/4$  каждый, и в каждом BVH содержится по 100 объектов.

**Уровень 3.** Пусть в нашем случае на этом уровне есть 10 BVH с радиусом  $r/8$  и 10 объектами в каждом.

В результате у нас получилось дерево, показанное на рис. 13.64. Теперь мы просто передаем это дерево функции пространственной отбраковки сфер, начиная с корневого узла (`Cull_OBJECT4DV2()`). Если из области видимости можно отбросить корневой узел, он

отбрасывается, и в этом случае ни один объект не является видимым — т.е. нам удастся свести проверку 10000 ограничивающих сфер к одной проверке. Но это очень маловероятно, и поэтому нам придется опуститься на следующий уровень дерева. Здесь ситуация становится интересней.

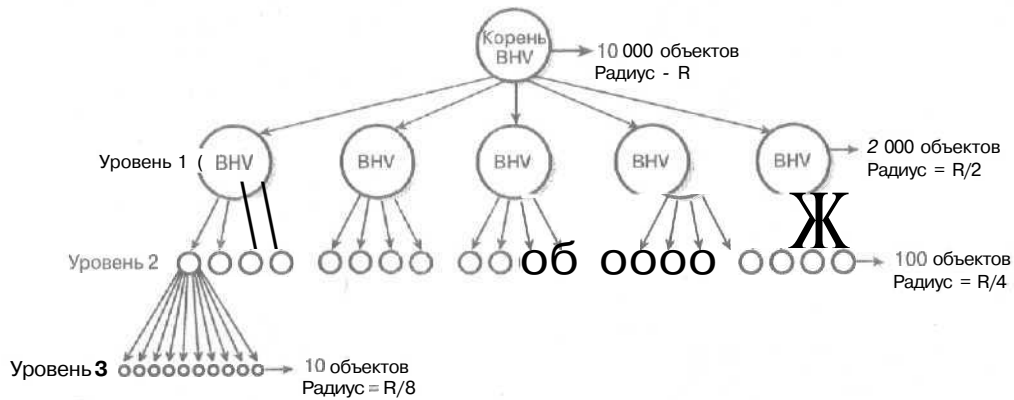


Рис. 13.64. Наглядный пример BVH

Корень имеет пять дочерних узлов, поэтому необходимо совершить обход этих пяти узлов в глубину или в ширину. Выберем метод обхода в ширину, т.е. будем обрабатывать каждый из пяти дочерних узлов и определять, отбрасывается ли данный BVH. Если это так, то все объекты данного узла помечаются как отброшенные, а его дочерние узлы помечаются как посещенные, соответственно, ни один из его дочерних узлов не посещается. Пусть, например, четыре из пяти BVH первого уровня отброшены в процессе обхода. Затем мы переходим на следующий уровень и обрабатываем дочерние узлы на следующем уровне — в этом случае их оказывается 20. Данный процесс продолжается до тех пор, пока не будут посещены все необходимые узлы BVH.

После завершения обхода все возможные объекты будут отброшены. В среднем речь идет о нескольких десятках тестов по сравнению с 10000; такая экономия весьма значительна. В этом и заключается польза BVH-методов.

Можно применить несколько очевидных оптимизаций. Во-первых, при обходе BVH-дерева вполне может оказаться, что некий объект находится в перекрывающихся узлах, поэтому он может быть отброшен на более высоком уровне, а затем отбрасывается еще раз (или производится такая попытка). В большинстве случаев это допустимо и зависит от процента перекрытия двух BVH. Иными словами, два BVH могут иметь общие объекты.

Второй существенной оптимизацией является счетчик, который просто подсчитывает число отброшенных объектов. Если значение счетчика становится равным общему числу возможных объектов, алгоритм обхода можно останавливать.

## Стратегии выбора

Последней составляющей картины является исходный алгоритм выбора BVH. В следующем разделе мы будем рассматривать детерминистический специальный случай BVH — октадеревья. Однако BVH общего вида могут быть по своей природе как детерминистическими, так и эвристическими. Таким образом, для выбора BVH можно вручную использовать специальный инструментарий, эвристический или детерминистиче-

ский алгоритмы. Все, что я **собираюсь** сделать — это просто предложить пару идей, которые я лично использовал в своих программах.

Кроме того, многие используют **BHV** для определения столкновений в реальном времени и отбраковки *движущихся* объектов. Это означает, что **BHV**-дерево должно генерироваться заново всякий раз при выходе объекта из узла, или для каждого кадра — в зависимости от конкретного случая. Поэтому обычно используются простые схемы с **небольшим** (3–8) количеством уровней. Следует помнить, что **BHV** — всего лишь вспомогательное средство, и нам не следует стремиться к совершенному результату. На первом месте всегда остается скорость выполнения.

### Разделяй и властвуй

Данная стратегия основана на согласованном **делении** трехмерного пространства на кубы, сферы и т.п., при этом размеры секторов одинаковы. Например, первым шагом всегда является включение пространства целиком в корневой уровень **BHV**. На втором **шаге** можно разделить пространство на кубы некоторого одинакового размера, **допустим** на  $16 \times 16 \times 16$  или  $4 \times 4 \times 4$  кубов, и в результате пространство будет равномерно поделено. Затем вокруг этих кубов описываются **BHV**. Любой сектор, не содержащий объектов, не включается в **BHV** данного уровня. Затем на каждом уровне **все составляющие** данный **уровень** **BHV** вновь разбиваются на кубы (или другие геометрические фигуры). Этот процесс показан на рис. 13.65а и 13.65б. Главным достоинством метода "разделяй и властвуй" является простота реализации. И вот так постепенно мы подходим к **октадеревьям** (в действительности мы и строим **октадеревья**!).

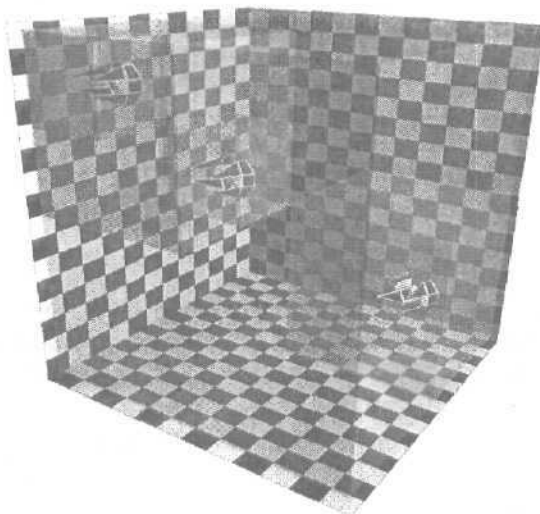


Рис. 13.65а. Секторизация трехмерного пространства, сплошная модель

### Кластеризация

Метод "разделяй и властвуй" легко реализуется, но он не слишком интеллектуален. Рассмотрим, например, рис. 13.66. Нетрудно заметить, что разбиение данного пространства на сектора является пустой тратой времени, поскольку можно сразу выделить три скопления объектов и использовать это в качестве отправной точки.

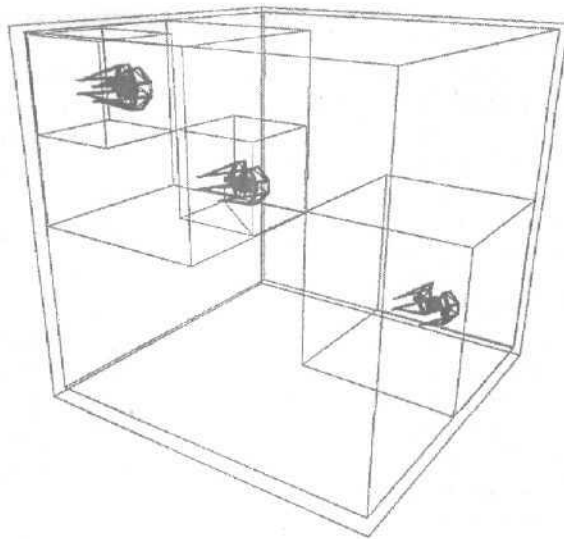


Рис. 13.656. Секторизация трехмерного пространства, каркасная модель

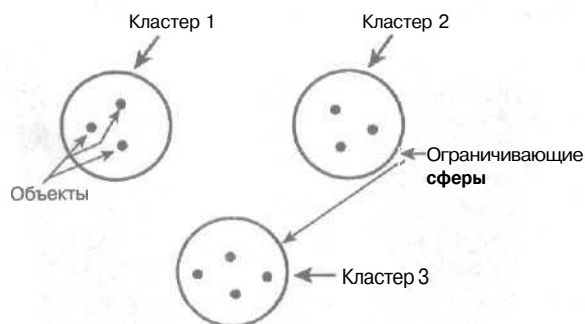


Рис. 13.66. В данном случае секторизация — напрасная трата времени

Таким образом, кластеризация работает с помощью эвристического алгоритма, основанного на случайном выборе отдельного объекта. Для этого объекта вычисляется среднее расстояние до всех других объектов. Объекты, которые находятся внутри определенного максимального диапазона, включаются в кластер. Это один из возможных способов кластеризации. Далее алгоритм продолжает свою работу, выбирая следующий случайный начальный объект для кластера и производит аналогичные действия.

Этот процесс продолжается до тех пор, пока не будет выполнено некое условие остановки (например, выбрано  $p$  начальных объектов кластеров или какое-то подобное). После этого используется наилучший начальный объект и соответствующий кластер, и процесс продолжается. На рис. 13.67 очень упрощенно показан данный метод для шести объектов. Каждый объект проверяется в качестве потенциального начального объекта кластера, но во время первого прохода выясняется, что наилучшим начальным объектом является первый, поскольку среднее расстояние от него до объектов его кла-

стера минимально. Таким образом, создается первый BNV, в качестве центра которого используется объект 1. Затем из оставшихся объектов создается следующий BNV; при этом его также следует заполнить по возможности более плотно.

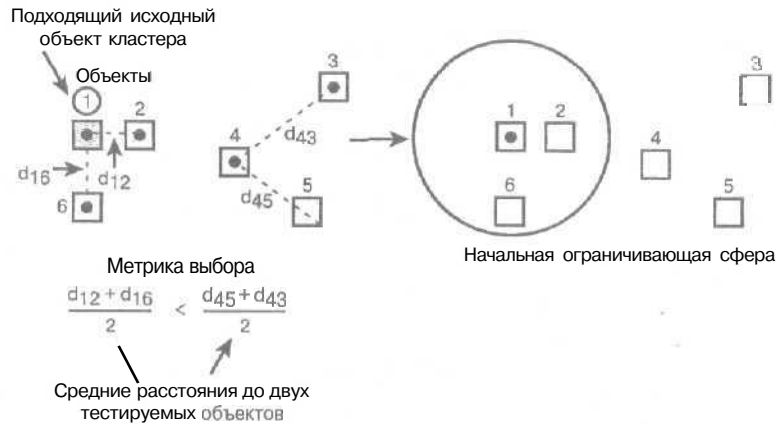


Рис. 13.67. Начальный выбор объекта основан на минимальном среднем расстоянии до всех его соседей

## Реализация BNV

В качестве примера использования BNV я написал демонстрационную программу, которая создает трехмерное пространство, загружает несколько сотен объектов и случайным образом размещает их в пространстве. Затем для группировки объектов на каждом уровне BNV применяется алгоритм "разделяй и властвуй". Алгоритм охватывает  $p$  уровней, где  $p$  задается программно. Функция, выполняющая анализ BNV, специально разработана для данного способа представления объектов. В большинстве других демонстрационных программ со множеством объектов загружается единственный объект типа OBJECT4DV2, а затем определяется массив, содержащий упорядоченное множество объектов, каждый из которых указывает на единственный корневой объект. Для работы с BNV мы еще раз воспользуемся этим методом, т.е. создадим массив виртуальных объектов, а затем визуализируем массив с использованием положений входящих в него объектов (рис. 13.68).

Тип контейнера главного объекта представлен ниже.

```
typedef struct OBJ_CONTAINERV1_TYP
{
    int state;    // Состояние объекта
    int attr;    // Атрибуты объекта
    POINT4D pos; // Положение объекта
    OBJECT4DV2_PTR obj; // Указатель на объект
} OBJ_CONTAINERV1, * OBJ_CONTAINERV1_PTR;
```

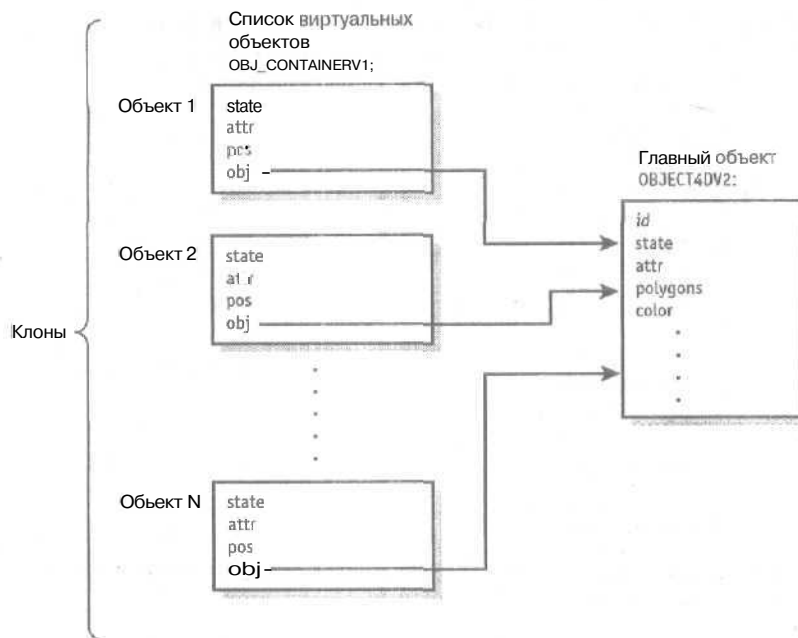


Рис. 13.68. Клонирование главного объекта

В качестве узла **BHV** я использовал ранее определенный тип **BHV\_NODEV1**. В процессе работы с **BHV** участвуют только две функции: одна из них создает **BHV**-дерево, а вторая отбрасывает узлы **BHV** из области видимости. Функция для создания **BHV** имеет следующий вид.

```
void BHV_Build_Tree(
    BHV_NODEV1_PTR bhv_tree,           // Создаваемое дерево
    OBJ_CONTAINERV1_PTR bhv_objects,    // Указатель на объекты
                                        // начальной сцены
    int num_objects,                   // Количество объектов
                                        // исходной сцены
    int level,                         // Уровень рекурсии
    int num_divisions,                 // Количество делений
                                        // на один уровень
    int universe_radius)               // Начальный размер
                                        // окружения
{
    // Данная функция создает BHV-дерево, используя для
    // объединения объектов в группы алгоритм "разделяй и
    // властвуй"

    Write_Error("\nEntering BHV function...");

    // Создание корневого узла?
    if (level == 0)
    {
        Write_Error("\nlevel = 0");
    }
}
```

```

// Позиция (0,0,0)
bhv_tree->pos.x = 0;
bhv_tree->pos.y = 0;
bhv_tree->pos.z = 0;
bhv_tree->pos.w = 1;

// Установка радиуса узла равным максимальному
bhv_tree->radius.x = universe_radius;
bhv_tree->radius.y = universe_radius;
bhv_tree->radius.z = universe_radius;
bhv_tree->radius.w = 1;

Write_Error("\nnode pos[%f, %f, %f], r[%f, %f, %f]",
    bhv_tree->pos.x, bhv_tree->pos.y,
    bhv_tree->pos.z,
    bhv_tree->radius.x, bhv_tree->radius.y,
    bhv_tree->radius.z);

// Создание корня путем добавления к нему всех
// объектов
for (int index = 0; index < num_objects; index++)
{
    // Проверка, не отброшен ли объект
    if (!(bhv_objects[index].state &
        OBJECT4DV2_STATE_CULLED))
    {
        bhv_tree->objects[bhv_tree->num_objects++] =
            (OBJ_CONTAINERV1_PTR)&bhv_objects[index];
    } // if
} // for index

// Все объекты включены в корневой узел, и параметр
// num_objects устанавливается равным числу объектов

// Деблокирование узла
bhv_tree->state = 1;
bhv_tree->attr = 0;

// Установка всех ссылок равными NULL
for (int ilink = 0; ilink < MAX_BHV_PER_NODE; ilink++)
    bhv_tree->links[ilink] = NULL;

// Установка количества объектов
bhv_tree->num_objects = num_objects;

Write_Error("\nInserted %d objects into root node",
    bhv_tree->num_objects);
Write_Error("\nMaking recursive call with "
    "root node...");

// Теперь можно рекурсивно строить остальную часть
// дерева

```

```

    BHV_Build_Tree(bhv_tree,
        bhv_objects,
        num_objects,
        1,
        num_divisions,
        universe_radius);
} // if
else
{
    Write_Error("\nEntering Level - %d > 0 block, "
        "number of objects = %d",
        level, bhv_tree->num_objects);

    // Проверка условия завершения
    if (bhv_tree->num_objects <= MIN_OBJECTS_PER_BHV_CELL)
        return;

    // Поарение дочернего узла (сложная часть). Нужно
    // взять текущий узел и расщепить его на несколько
    // дочерних узлов, затем создать bhv для каждого
    // дочернего узла и включить объекты а каждый
    // дочерний узел, после чего для каждого дочернего
    // узла снова рекурсивно вызвать функцию
    // построения...

    // Создание временных трехмерных ячеек для
    // отслеживания
    BHV_CELL cells[MAX_BHV_CELL_DIVISIONS]
        [MAX_BHV_CELL_DIVISIONS]
        [MAX_BHV_CELL_DIVISIONS];

    // Поиск исходной точки ограничивающего объема на
    // основании заданных радиуса и центра
    int x0 = bhv_tree->pos.x - bhv_tree->radius.x;
    int y0 = bhv_tree->pos.y - bhv_tree->radius.y;
    int z0 = bhv_tree->pos.z - bhv_tree->radius.z;

    // Вычисление размеров ячейки вдоль осей x,y,z
    float cell_size_x = 2*bhv_tree->radius.x /
        (float)num_divisions;
    float cell_size_y = 2*bhv_tree->radius.y /
        (float)num_divisions;
    float cell_size_z = 2*bhv_tree->radius.z /
        (float)num_divisions;

    Write_Error("\ncell pos=(%d, %d, %d) "
        "size=(%f, %f, %f)",
        x0,y0,z0, cell_size_x,
        cell_size_y, cell_size_z);

    int cell_x, cell_y,
        cell_z; // Используется для задания положения
        // ячейки в трехмерной матрице

```

```

// Очистка памяти для ячеек
memset(cells, 0, sizeof(cells));

// Разбиение пространства на num_divisions частей
// (это число должно быть < MAX_BHV_CELL_DIVISIONS),
// а затем сортировка центров каждого объекта в
// ячейки трехмерной матрицы сортировки
for(int obj_index = 0;
    obj_index < bhv_tree->num_objects; obj_index++)
{
    // Вычисление позиции ячейки во временной
    // матрице сортировки
    cell_x = (bhv_tree->objects[obj_index]->pos.x
        - x0)/cell_size_x;
    cell_y = (bhv_tree->objects[obj_index]->pos.y
        - y0)/cell_size_y;
    cell_z = (bhv_tree->objects[obj_index]->pos.z
        - z0)/cell_size_z;

    // Включение объекта в список
    cells[cell_x][cell_y][cell_z]
        .obj_list[cells[cell_x][cell_y][cell_z]
            .num_objects] = bhv_tree->objects[obj_index];

    Write_Error("\ninserting object %d located at "
        "(%f, %f, %f) into cell "
        "(%d, %d, %d)",
        obj_index,
        bhv_tree->objects[obj_index]->pos.x,
        bhv_tree->objects[obj_index]->pos.y,
        bhv_tree->objects[obj_index]->pos.z,
        cell_x, cell_y, cell_z);

    // Увеличение числа объектов в данной ячейке
    if (++cells[cell_x][cell_y][cell_z].num_objects
        >= MAX_OBJECTS_PER_BHV_CELL)
        cells[cell_x][cell_y][cell_z].num_objects =
            MAX_OBJECTS_PER_BHV_CELL-1;
} // for obj_index

Write_Error("\nEntering sorting section...");

// Теперь трехмерная матрица сортировки содержит всю
// необходимую информацию. Следующий шаг состоит в
// создании BHV-узла для каждой непустой ячейки

for(int icell_x = 0; icell_x < num_divisions;
    icell_x++)
{
    for(int icell_y = 0; icell_y < num_divisions;
        icell_y++)
    {

```

```

for(int icell_z = 0;
    icell_z < num_divisions; icell_z++)
{
    // Есть ли объекты в данном узле?
    if (cells[icell_x][icell_y][icell_z]
        .num_objects > 0)
    {
        Write_Error("\nCell %d, %d, %d "
            "contains %d objects",
            icell_x, icell_y, icell_z,
            cells[icell_x][icell_y][icell_z]
            .num_objects);

        Write_Error(
            "\nCreating child node...");

        // Создание узла и установка ссылки
        // на него
        bhv_tree->links
            [bhv_tree->num_children] =
            (BHV_NODEV1_PTR)malloc(
                sizeof(BHV_NODEV1));

        // Обнулить узел
        memset(bhv_tree->links[
            bhv_tree->num_children],
            0, sizeof(BHV_NODEV1));

        // Установить данный узел
        BHV_NODEV1_PTR curr_node =
            bhv_tree->links[
                bhv_tree->num_children];

        // Позиция
        curr_node->pos.x =
            (icell_x*cell_size_x +
                cell_size_x/2) + x0;
        curr_node->pos.y =
            (icell_y*cell_size_y +
                cell_size_y/2) + y0;
        curr_node->pos.z =
            (icell_z*cell_size_z +
                cell_size_z/2) + z0;
        curr_node->pos.w = 1;

        // Радиус равен cell_size/2
        curr_node->radius.x = cell_size_x/2;
        curr_node->radius.y = cell_size_y/2;
        curr_node->radius.z = cell_size_z/2;
        curr_node->radius.w = 1;

        // Указание числа объектов
        curr_node->num_objects =

```

```

        cells[icell_x][icell_y][icell_z]
        .num_objects;

// Указание числа дочерних узлов
curr_node->num_children = 0;

// Указание состояния и атрибутов
// Разблокировать узел
curr_node->state = 1;
curr_node->attr = 0;

// Включить каждый объект в список
// объектов данного узла
for (int icell_index = 0;
     icell_index <
     curr_node->num_objects;
     icell_index++)
{
    curr_node->objects[icell_index] =
    cells[icell_x][icell_y][icell_z]
    .obj_list[icell_index];
} // for icell_index

Write_Error("\nChild node pos="
            "(%f, %f, %f), r=(%f, %f, %f)",
            curr_node->pos.x,
            curr_node->pos.y,
            curr_node->pos.z,
            curr_node->radius.x,
            curr_node->radius.y,
            curr_node->radius.z);

// Увеличить число дочерних узлов
// родительского узла
bhv_tree->num_children++;

} // if

} // for icell_z

} // for icell_y

} // for icell_x

Write_Error("\nParent has %d children..",
            bhv_tree->num_children);

// Для каждого дочернего узла строим BHV
for (int inode = 0; inode < bhv_tree->num_children;
     inode++)
{
    Write_Error("\nfor Level %d, creating "
                "child %d", level, inode);

```

```

        BHV_Build_Tree(bhv_tree->links[inode],
            NULL, // Сейчас не используется
            NULL, // Сейчас не используется
            level+1,
            num_divisions,
            universe_radius);

    } // if

} // else level > 0

Write_Error("\nExiting BHV...level = %d", level);

} // BHV_Build_Tree

Далее представлена функция пространственной отбраковки.

int BHV_FrustumCull(
    BHV_NODEV1_PTR bhv_tree, // Корень BHV
    CAM4DV1_PTR cam,         // Камера, относительно
                             // которой производится отбор
    int cull_flags)           // Плоскости отсечения
{
    // ЗАМЕЧАНИЕ. Функция основана на использовании матриц.
    // Данная функция отбрасывает BHV из области видимости,
    // используя переданную информацию о точке наблюдения и
    // флагах cull_flags, определяющих, по каким осям
    // следует производить отбор— x, y, z или всем сразу,
    // что определяется путем побитовой операции ИЛИ над
    // флагами. При отбраковке BHV информация о состоянии в
    // каждом узле модифицируется таким образом, чтобы ее
    // могла использовать функция визуализации

    // Проверка корректности BHV и точки наблюдения
    if (!bhv_tree || !cam)
        return (0);

    // Необходимо обойти дерево сверху вниз

    // Шаг 1: преобразование центра ограничивающей сферы
    // узлов в пространство камеры

    POINT4D sphere_pos; // Хранит результат преобразование
                        // центра ограничивающей сферы

    // Преобразование точки
    Mat_Mul_VECTOR4D_4X4(&bhv_tree->pos, &cam->mcam,
        &sphere_pos);

    // Шаг 2: удаление объекта на основании значений
    // флагов cull_flags
    if (cull_flags & CULL_OBJECT_Z_PLANE)
        i

```

```

// Отбор основан на отсекающих плоскостях,
// перпендикулярных Z
if (((sphere_pos.z - bhv_tree->radius.z) >
    cam->far_clip_z) ||
    ((sphere_pos.z + bhv_tree->radius.z) <
    cam->near_clip_z))
{
    // Отбрасывается весь данный узел. Нужно
    // установить флаг culled для каждого объекта
    for(int iobject = 0;
        iobject < bhv_tree->num_objects; iobject++)
    {
        SET_BIT(bhv_tree->objects[iobject]->state,
            OBJECT4DV2_STATE_CULLED);
    } // for iobject

    // Узел был посещен и отброшен
    bhv_nodes_visited++;

    return(1);
} // if
} // if

if (cull_flags & CULL_OBJECT_X_PLANE)
{
    // Отбор, основанный на отсекающих плоскостях,
    // перпендикулярных X. Можно использовать уравнения
    // плоскостей, но простые подобные треугольники
    // проще, поскольку в действительности это двумерная
    // задача. Если обзор составляет 90°, задача
    // тривиальна, однако предположим, что это не так

    // Проверка положения правой и левой отсекающих
    // плоскостей относительно самой левой и самой
    // правой точек ограничивающей сферы
    float z_test = (0.5)*cam->viewplane_width*
        sphere_pos.z/cam->view_dist;

    if (((sphere_pos.x - bhv_tree->radius.x) >
        z_test) ||
        ((sphere_pos.x + bhv_tree->radius.x) <
        -z_test)) // Обратите внимание на изменение
        // знака
    {
        // Данный узел отбрасывается целиком. Необходимо
        // установить соответствующий флаг для каждого
        // объекта
        for(int iobject = 0;
            iobject < bhv_tree->num_objects; iobject++)
        {
            SET_BIT(bhv_tree->objects[iobject]->state,
                OBJECT4DV2_STATE_CULLED);
        }
    }
}

```

```

    } // for iobject

    // Данный узел посещен и отброшен
    bhv_nodes_visited++;

    return (1);
} // if
} // if

if (cull_flags & CULL_OBJECT_Y_PLANE)
{
    // Отбор, основанный на отсекающих плоскостях,
    // перпендикулярных у. Можно использовать уравнения
    // плоскостей, но простые подобные треугольники
    // проще, поскольку в действительности это двумерная
    // задача. Если обзор составляет 90°, задача
    // тривиальна, однако предположим, что это не так

    // Проверка положений верхней и нижней отсекающих
    // плоскостей относительно самой верхней и самой
    // нижней точек ограничивающей сферы

    float z_test = (0.5)*cam->viewplane_height*
        sphere_pos.z/cam->view_dist;
    if (((sphere_pos.y - bhv_tree->radius.y) >
        z_test) ||
        ((sphere_pos.y + bhv_tree->radius.y) <
        -z_test)) // Обратите внимание на изменение
        // знака
    {
        // Данный узел отбрасывается целиком. Необходимо
        // установить соответствующий флаг для каждого
        // объекта
        for(int iobject = 0;
            iobject < bhv_tree->num_objects; iobject++)
        {
            SET_BIT(bhv_tree->objects[iobject]->state,
                OBJECT4DV2_STATE_CULLED);
        } // for iobject

        // Данный узел посещен и отброшен
        bhv_nodes_visited++;

        return(1);
    } // if
} // if

// В этой точке мы пришли к выводу, что данный BHV-узел
// слишком велик, чтобы быть отброшенным целиком,
// поэтому необходимо совершить обход его дочерних узлов
// и выяснить, нельзя ли отбросить их

for (int ichild = 0; ichild < bhv_tree->num_children;

```

```

        ichild++)
    i
    // Рекурсивный вызов функции...
    BHV_FrustumCull(bhv_tree->links[ichild],
        cam, cull_flags);

    // Здесь можно прибегнуть к оптимизации: отслеживать
    // общее число отброшенных объектов и завершить
    // процесс, если отброшены все объекты...

} // for ichild
return(0);
} // BHV_FrustumCull

```

Функция `BHV_FrustumCull()` совершает обход **BHV-дерева**, как описано выше, и применяет алгоритм пространственной отбраковки **ограничивающих сфер** из функции `Cull_OBJECT4DV2()`. По существу мы используем центр **BHV** и его радиус для создания **ограничивающей сферы**, а затем проверяем, находится ли данная ограничивающая **сфера** в области обзора.

#### НА ЗАМЕТКУ

Все будет в порядке, если ограничивающий объем является сферой; но если по умолчанию ограничивающий объем является кубом, то в некоторых случаях вы будете видеть ранее отброшенные объекты. Это легко исправить: необходимо написать программу отбраковки, поддерживающую кубы. Я предлагаю вам сделать это самостоятельно, поскольку в реальных играх вам могут понадобиться различные формы объемов. Сферы же удобнее всего использовать для демонстрационных целей.

#### СОВЕТ

Реализовать отбор кубов относительно просто: вместо нахождения шести точек касания сферы можно использовать **восемь** вершин ограничивающего куба.

По окончании работы программы `BHV_FrustumCull()` массив объектов обновляется с учетом поля состояния (установлен бит отбраковки или нет). Затем в цикле визуализации объектов производится простая проверка массива объектов. Отброшенные объекты игнорируются; остальные объекты подвергаются обычной обработке, которая включает в себя стандартный отбор с **помощью** ограничивающих сфер.

На рис. 13.69 показана копия экрана демонстрационной программы `DEMO113_5.EXE|CPP`. Она не выглядит особо впечатляюще, поскольку основные действия происходят как раз вне экрана, но выводимая информация о состоянии программы представляет особый интерес. В ней представлено число отброшенных объектов, число посещенных **BHV-узлов** и количество объектов, отправленных в цикл визуализации (не отброшенных **методом BHV**). С **помощью** клавиши <V> можно переключать режим использования **BHV**.

#### НА ЗАМЕТКУ

Чтобы самостоятельно скомпилировать демонстрационную программу, вам понадобятся исходный файл `DEMO113_5.CPP`, а также модули `I3LIB1-11.CPP|H` и библиотечные файлы DirectX. Кроме того, вам понадобится файл ресурсов `DEMO113_5.RC`.

## Октадеревья

Октадеревья являются частным случаем только что рассмотренной системы **BHV**. К созданию октадера приводит выбор **числа делений** в функции `BHV_Build_Tree()`, **равного** двум.

Итак, **октадеревя** — это просто **BHV-дерево**, выровненное по осям координат, узлы которого имеют дочерние узлы. Все узлы одного уровня имеют одинаковые размеры и

кубическую форму Система **BHV** допускает существование некубических ограничивающих объектов, поскольку радиус представлен вектором, а в каждом узле имеется слово-атрибут, которое потенциально может описывать форму ограничивающего объема.

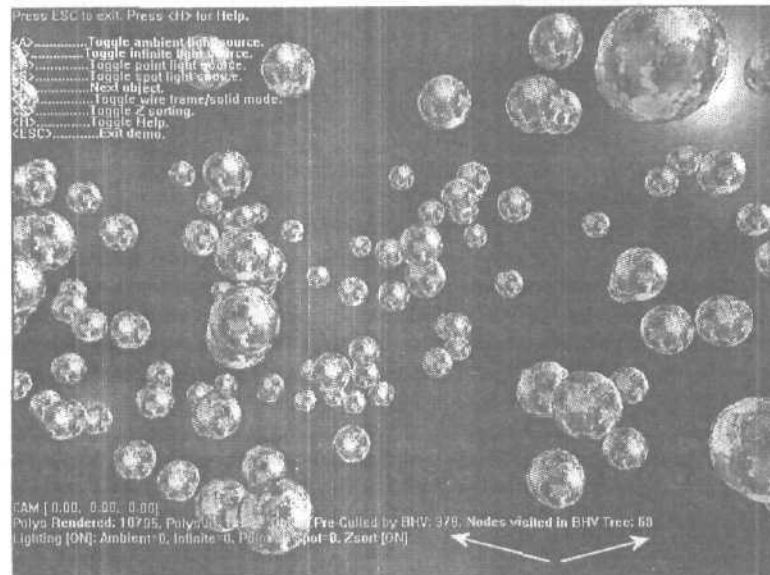


Рис. 13.69. Копия экрана демонстрационной программы, работающей с **BHV**

Таким образом, мы уже построили систему **октадеревьев** и ничего нового писать не нужно. Однако есть некоторые вопросы, на которых хотелось бы остановиться подробнее.

**Октадеревья** (как и **BHV**) могут работать как на уровне объектов, так и многоугольников. В рассматривавшемся случае мы работали с объектами, а не многоугольниками (многоугольники мы использовали в демонстрации **BSP**), поскольку **октадеревья** и **BHV-деревья** наиболее полезны именно при работе с большими открытыми пространствами с тысячами объектов.

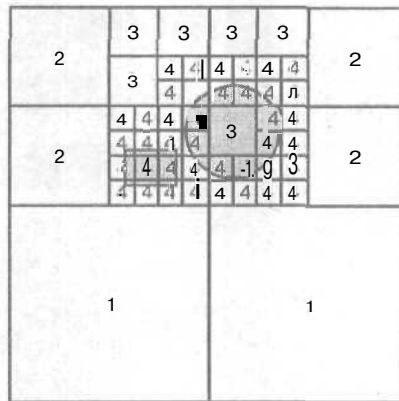
В трехмерном пространстве узел **октадера** всегда имеет до восьми дочерних узлов одинакового размера, однако не существует правила, которое требует, чтобы все дочерние узлы входили в **октадерев**.

Так, на рис. 13.70 показано несколько итераций алгоритма создания **BHV-октадера** и соответствующее разбиение пространства (двумерная проекция с нумерацией, показывающей уровень итерации). Однако не все пространство разбивается равномерно: дальнейшее разбиение производится только для тех узлов, в которых содержатся многоугольники или объекты. Такое решение позволяет экономить память. При желании можно создавать на каждой итерации все восемь узлов и ничего не помещать в них, но я предпочитаю вовремя останавливаться и допускать наличие пустых (**NULL**) октантов.

При создании **октадера**, как и **BHV** общего вида, существует ряд условий остановки. Можно останавливаться при достижении определенного размера октанта, определенной глубины рекурсии или тогда, когда в каждом узле содержится некое минимальное число **многоугольников**. Выбор остается за вами.

В заключение еще раз подчеркиваем, что **октадеревья** являются не более чем частным случаем **BHV**. Разработанный нами алгоритм будет создавать **октадерев**, если задать количество делений равным двум, например, **следующим** образом.

```
BHV_Build_Tree(&bhv_tree,
scene_objects,
NUM_SCENE_OBJECTS,
0,
2);
```



ft/c. 13.7ft Двумерное представление  
октадеревя

Пространственный отбор осуществляется точно **так же**. Единственной причиной использования октадеревя вместо более сложной схемы деления (например, с 10 делениями на уровень) является то, что восемь октантов по сути соответствуют восьми октантам трехмерной системы координат, которые имеют ряд полезных свойств и с которыми легче работать.

## Отбор с учетом препятствий

Последняя тема данной главы вновь связана с выводом видимых объектов. Мы уже видели, что с помощью технологии PVS можно генерировать списки многоугольников для любой точки обзора в игре, а затем обращаться к этим спискам, чтобы определить, какие многоугольники следует учитывать при создании определенного кадра. Но что, если вы не используете PVS или этот метод неосуществим или бесполезен в силу особенностей разрабатываемой вами игры?

Существует еще один класс методов, использующий противоположный подход к проблеме, а именно — *отбор с учетом препятствий* (occlusion culling). Этот метод может использоваться, например, когда сложно получить статическое решение проблемы видимости (например, игровое пространство быстро меняется, и использование PVS, BSP и октадеревя *нечелесообразно*), но при этом в игровом пространстве существуют некие ключевые геометрические элементы, которые настолько велики, что закрывают большую часть сцены.

Рассмотрим, например, копию экрана игры *Unreal 2003* на рис. 13.71, а затем посмотрим на рис. 13.72. Второй рисунок получен после простого поворота игрока. Все, что произошло, — перед игроком оказался огромный объект, который закрывает практически все поле зрения. PVS здесь не поможет, порталы также бесполезны, однако понятно, что должен существовать некий метод, который можно использовать в этом простейшем случае, чтобы не визуализировать и даже не рассматривать **многоугольники, заслоненные** этим объектом. В этом и состоит основная идея отбраковки с учетом препятствий.

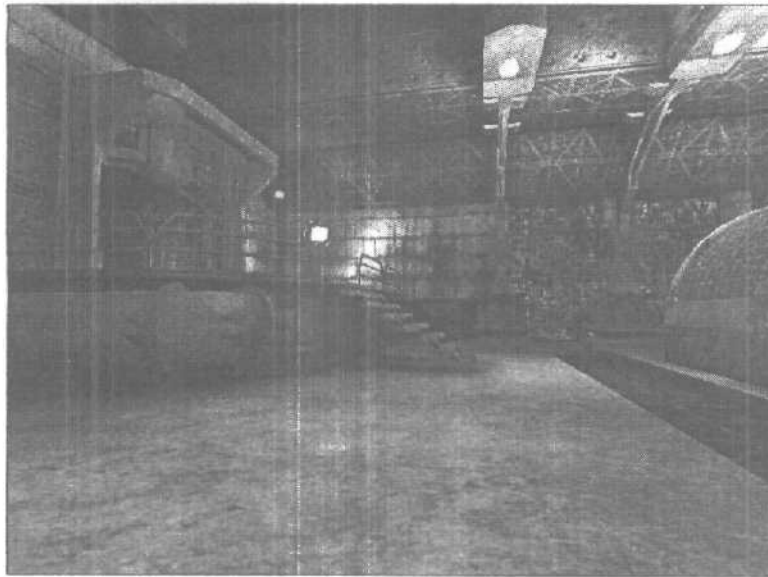


Рис. 13.71. Открытый вид игрового пространства Unreal 2003



Рис. 13.72. Вид после поворота

## Перекрытые пространства

Рассмотрим простой случай из практики игр на открытом пространстве с городскими пейзажами. На рис. 13.73а представлен вид сверху некого города (на самом деле — Хьюстона), где маленькие прямоугольники представляют **постройки**. Никаких комнат, порталов и т.д. Можно было бы использовать PVS, однако без ограничивающей внутренней геометрии, чтобы построить PVS для каждой точки наблюдения, нам

придется считать ограничивающей ячейкой область видимости. Так что забудем об этом, и сконцентрируемся на очевидном: на расположении точки наблюдения на рис. 13.73а и на области видимости вокруг нее. Перед точкой наблюдения находится большое темное здание. Теперь посмотрим на рис. 13.73б — это здание закрывает 90% потенциально видимых многоугольников!

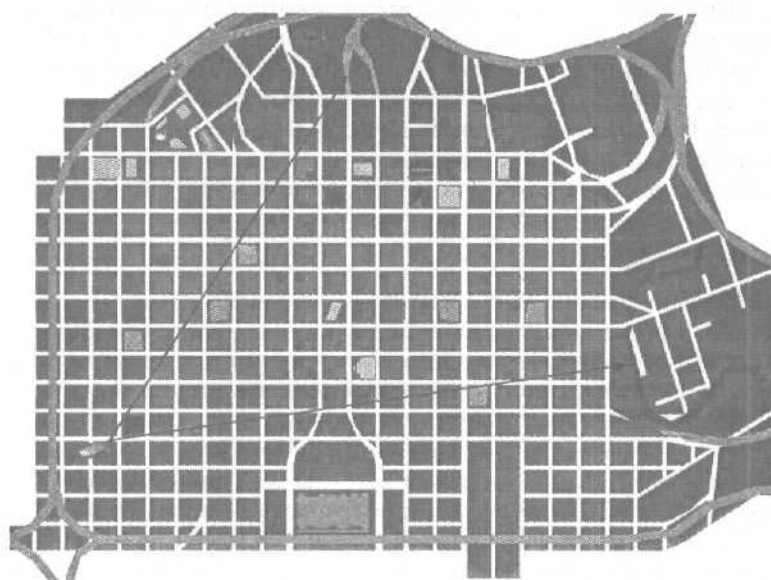


Рис. 13.73а. Городской пейзаж для демонстрации перекрытия

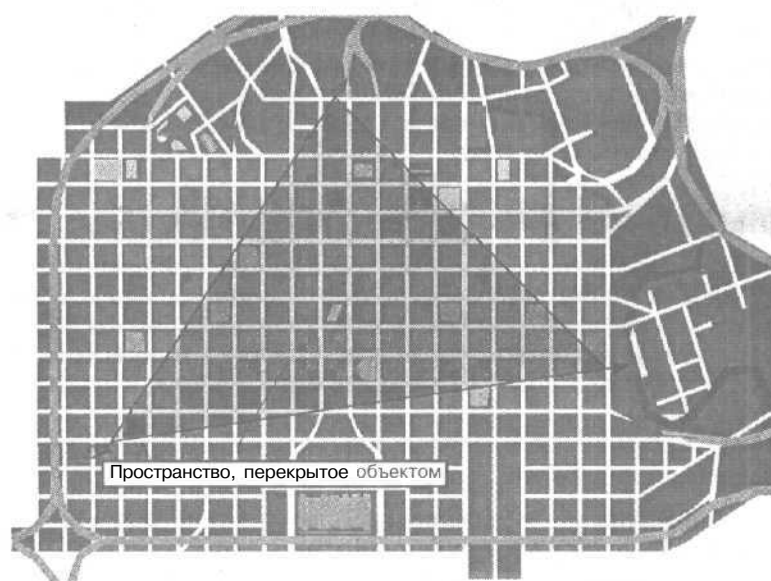


Рис. 13.73б. Тот же пейзаж после визуализации тени объема окклюзии

Таким образом, можно решить поставленную задачу, если создать перекрытое пространство. Все находящиеся в нем объекты оказываются невидимы. Очевидно, что создание **ограничивающих** объемов такого вида — непростая задача. Существует множество алгоритмов нахождения силуэта трехмерного объекта, по которому можно **построить** необходимое количество плоскостей, определяющих перекрытую область.

Все, оказавшееся внутри данной области, отсекается. **Можно** развить эту идею и при наличии нескольких объектов-преград создать несколько перекрытых областей. При этом, однако, возникает проблема возрастания сложности определения: находится ли определенный многоугольник в одной из перекрытых областей? А на это требуется больше времени, чем для простого вывода многоугольника. Однако совершенно необязательно использовать в качестве наименьшей единицы отбраковки многоугольники. Можно рассмотреть перекрытие объектов, BSP-деревьев, ячеек **октадеревьев** и т.д.

## Выбор преград

Для осуществления отбраковки перекрытием необходим некий метод, позволяющий выбирать, какие объекты являются потенциальными преградами для поля обзора (этот метод, кстати, может использовать определенные **предвычисления** силуэтов-преград). Существует ряд стратегий, используемых для выбора преград, но все их можно отнести к одной из двух категорий: предварительный выбор до начала **игры** и выбор в процессе игры. При выборе преград до начала игры необходимо отметить определенные объекты как потенциальные преграды. Помечаются крупные объекты или объекты, которые могут длительно загораживать точку обзора игрока в процессе игры. Кроме того, множество **потенциальных** преград может выбираться на основе секторов.

При выборе преград в процессе игры выполняется, как минимум, отметка потенциальных преград. В процессе выполнения система определяет, включать ли данную преграду в список, превращает ее в область перекрытия и выполняет соответствующий отбор. Используемые эвристики основываются на размерах преград. Это означает, что в качестве потенциальных преград помечаются только крупные движущиеся объекты или крупные объекты игрового мира. Затем в ходе выполнения система вычисляет спроецированное экранное пространство такого объекта и принимает решение, стоит ли использовать его в качестве преграды. **Воспользовавшись** данной идеей, я разработал гибридный метод, которым часто пользуюсь.

## Гибридный метод выбора преград

Преграды можно выбирать как до начала игры или еще на этапе моделирования, так и в процессе игры из множества потенциальных преград; их также можно выбирать "налету", путем вычисления объема крупных проецируемых объектов (или многоугольников) и принятия решения о том, стоит ли использовать данный объект в качестве преграды. Чтобы встроить все эти методы в процесс моделирования и конвейер визуализации, придется достаточно много поработать. Однако существует ряд простых способов выбора преград, основанных на многоугольниках или на объектах. Обычно такой процесс состоит из **следующих** этапов.

1. Для каждого потенциально видимого объекта из области обзора следует выбрать **п** ближайших объектов с **наибольшими** размерами.
2. Затем надо спроецировать данные объекты или геометрические элементы в экранное пространство и посмотреть на площадь полученных проекций. После этого выбрать  $m \leq p$  объектов, занимающих наибольшее экранное пространство, затем

вычислить ограничивающие прямоугольники, содержащиеся внутри каждой области перекрытия (хотя при этом некоторые потенциальные преграды теряются, проецировать ограничивающие прямоугольники гораздо легче, чем произвольные многоугольники или ячейки).

3. Теперь надо пройти по всем объектам и многоугольникам в областях перекрытия, удаляя все закрытые геометрические элементы.

По сути, данный алгоритм достаточно прост и прямолинеен: осуществляется быстрый проход и выбирается несколько ближайших наиболее крупных объектов или геометрических элементов; предполагается, что они, вероятно, окажутся преградами. Затем их объемы/площади проецируются в экранное пространство, и выбираются те преграды, которые могут закрыть значительную часть области видимости. Однако вместо сложной геометрии в качестве основы построения областей перекрытия используются вписанные ограничивающие прямоугольники, полностью находящиеся внутри каждой преграды. Затем любой объект, более удаленный по оси  $z$ , чем плоскость преграды, и находящийся внутри перекрытого объема, исключается из рассмотрения. Этот метод лучше всего работает, когда имеется некая иерархия представления объектов в игровом мире, например, BSP-деревья, комнаты, октадеревья и т.п., тогда сразу удастся исключить из рассмотрения значительное количество объектов.

## Резюме

Мы рассмотрели ряд важных алгоритмов разбиения пространства, изучили их практическое применение и реализовали наиболее существенные из них. Этих знаний достаточно, чтобы создавать собственные алгоритмы определения видимости, примером чего может служить только что описанный гибридный метод. Изложенные в данной главе идеи принесут несомненную пользу при написании ваших собственных игр.

Однако прежде чем начать кодировать, убедитесь, что вы прочитали все, что только можно, о данном предмете, чтобы вам не пришлось вновь и вновь изобретать велосипед. Конечно, я часто советую попытаться все сделать самостоятельно, но только чтобы получить необходимый опыт. Если же у вас мало времени, то лучше ознакомиться с уже существующими разработками по данной теме.

И наконец, одно предостережение. Многие алгоритмы являются рекурсивными, поэтому берегите свое время и будьте очень внимательны, поскольку отладка рекурсии — это кошмар, поскольку отладка рекурсий — это кошмар, поскольку отладка рекурсий — это кошмар...

Stack Overflow: Instruction Pointer - 0x00fff564

Надеюсь, вы поняли, что я имел в виду? :-)



# ГЛАВА 14

## Освещение и тени

### В этой главе...

• Новый модуль игрового процессора	1264
• Введение и план игры	1264
• Упрощенная физика теней	1264
• Имитация теней с помощью проецирования изображений и макетов	1269
• Отображение тени на плоскую сетку	1290
• Введение в отображение освещения и кэширование поверхности	1296

В этой главе рассматривается отображение тени и света, что можно успешно использовать для придания большей реалистичности нашим сценам. Как обычно, мы в первую очередь рассматриваем программные методы. Вот о чем мы поговорим в данной главе:

- новый библиотечный модуль игрового процессора;
- введение и план игры;
- физика теней;
- моделирование теней с помощью проецирования изображений и макетов;
- отображение тени на плоскую сетку;
- отображение освещенности и кэширование поверхности;
- большое количество демонстрационных примеров!

## Новый модуль игрового процессора

В этой главе, как и в других, предоставлено достаточное количество кода, чтобы вынести его в отдельный библиотечный модуль — `T3DLIB12.CPP|H`. Таким образом, для компиляции программ из данной главы необходимы `.CPP`-файл программы, библиотечные файлы `DirectX` и новые библиотечные файлы:

`T3DLIB12.CPP` — исходный C/C++ код для добавления тени и света.

`T3DLIB12.H` — сопровождающий заголовочный файл.

Разумеется, необходимо также подключить файлы `T3DLIB1-11.CPP|H`.

## Введение и план игры

Нами уже было рассмотрено достаточное количество материала, касающегося проблем отображения **освещенности**. Например, мы использовали текстуры и освещение в главе 8, "Основы моделирования освещения и поверхностей тел". Что касается теней, то у нас уже есть код **растеризатора** на основе альфа-смешивания, который потенциально может быть использован для изображения теневых проекций. Таким образом, в этой главе будет *выполнена системная интеграция* того, что уже было изучено ранее, и добавлено несколько вещей, которые обеспечат написание демонстрационных программ. Но я уверен, что в целом вы сможете закончить каждую из моих функций, потому что вы уже знаете, как это сделать.

В любом случае, по моему мнению, лучше начать с краткого исследования теней с точки зрения физики, а затем реализовать несколько методов их создания (этих методов достаточно много, но в основном они хорошо работают при аппаратной реализации, так что я оставляю их для книг, описывающих `Direct3D/OpenGL`). После этого можно перейти к рассмотрению отображения освещения и кэширования поверхности (в очень упрощенном варианте вместо рассмотрения полной поддержки этого механизма). Я принял такое решение, потому что добавление общего отображения освещенности в наш игровой процессор потребует раз в десять больше кода, чем упрощенный пример. Кроме того, по моему мнению, простой пример более выразителен, чем длинная реализация.

Итак, с этого момента вы должны быть способны к созданию программного процессора, предоставляющего возможность **визуализации миров**, похожих на миры *Quake/Unreal*. А к концу главы в вашем распоряжении будет еще несколько специальных эффектов.

## Упрощенная физика теней

Как же образуются тени? И почему они имеют размытые или резкие границы?

На рис. 14.1 изображен цилиндр, опирающийся на плоскость, с точечным источником света, который находится выше этого цилиндра. Благодаря небольшому отдаленному освещению цилиндр видим со всех сторон, но цилиндр отбрасывает тень на горизонтальную плоскость, потому что он препятствует прохождению луча от точечного источника света на эту плоскость — это и есть тень. Однако если внимательнее посмотреть на изображение, то можно заметить интересный эффект второго порядка.

Анализируя непосредственно саму тень, можно увидеть, что она в середине имеет сплошную темную область (*полную тень*), но по краям, где тень обрывается, видна более яркая область (или менее затененная). Эта область называется *полутенью*. Также следует

обратить внимание на то, что полутень отчетливо видна только в дальних участках тени, но не у основания цилиндра.

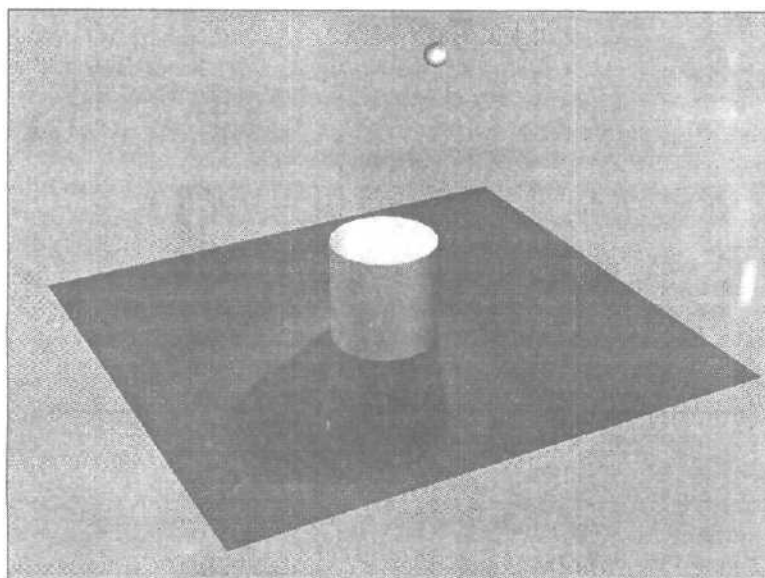


Рис. 14.1. Простая тень, отбрасываемая цилиндром

При осмотре основания цилиндра и тени в этой области можно увидеть, что тень очень резкая и интенсивная. Это и есть те детали, которые мы хотим имитировать, моделировать или реализовать иным способом в нашем процессоре. Давайте рассмотрим, что происходит на уровне фотонов.

## Движение фотонов и вычислительная интенсивность

Посмотрите на рис. 14.2а. На нем изображен вид сбоку нашего цилиндра с единственным точечным источником освещения. При отслеживании траектории лучей от источника **освещения** до пересечения с горизонтальной плоскостью можно увидеть, что результаты тривиальны. Единственный точечный источник света **освещает** все, что не находится в затененной области (области, закрытой цилиндром). Поэтому, как показано на рис. 14.2б, мы получаем **тень** с четкими границами.

А что будет в том случае, когда в качестве источника освещения выступает не точка, а маленькая сфера? Этот случай проиллюстрирован на рис. 14.3. В ситуации, представленной на рис. 14.3, заслуживает внимания тот факт, что для любой точки  $p_i$ , расположенной на оси  $x$  тени, отбрасываемой цилиндром, интенсивность света является различной для каждой позиции  $x$  и прямо пропорциональна площади проекции, определяемой этой точкой и двумя точками, пересекающими сферу источника света.

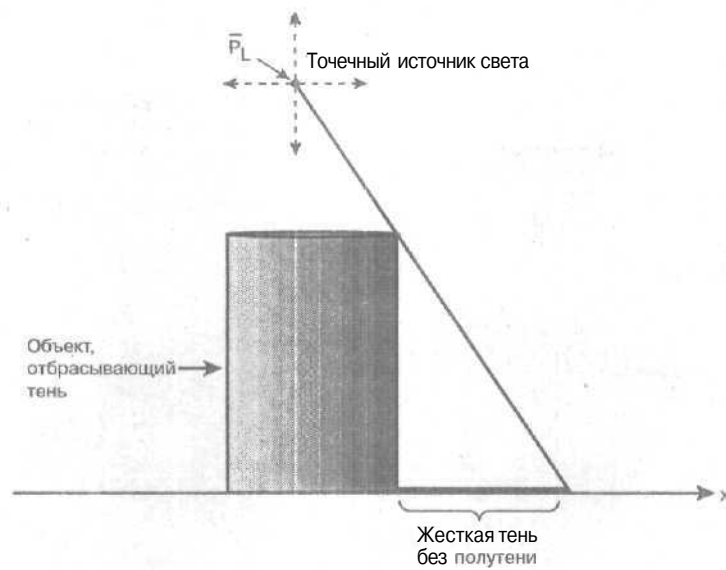


Рис. 14.2a. Трассировка лучей точечного источника света

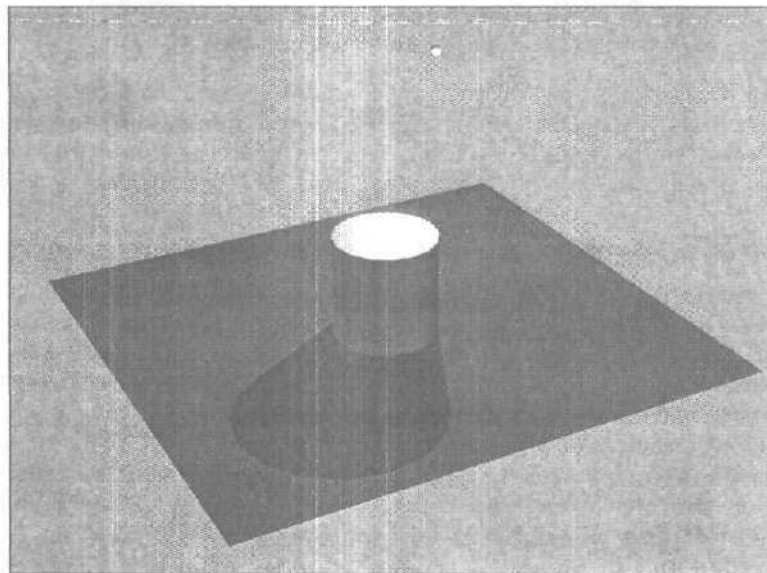


Рис. 14.2b. Тень от точечного источника света

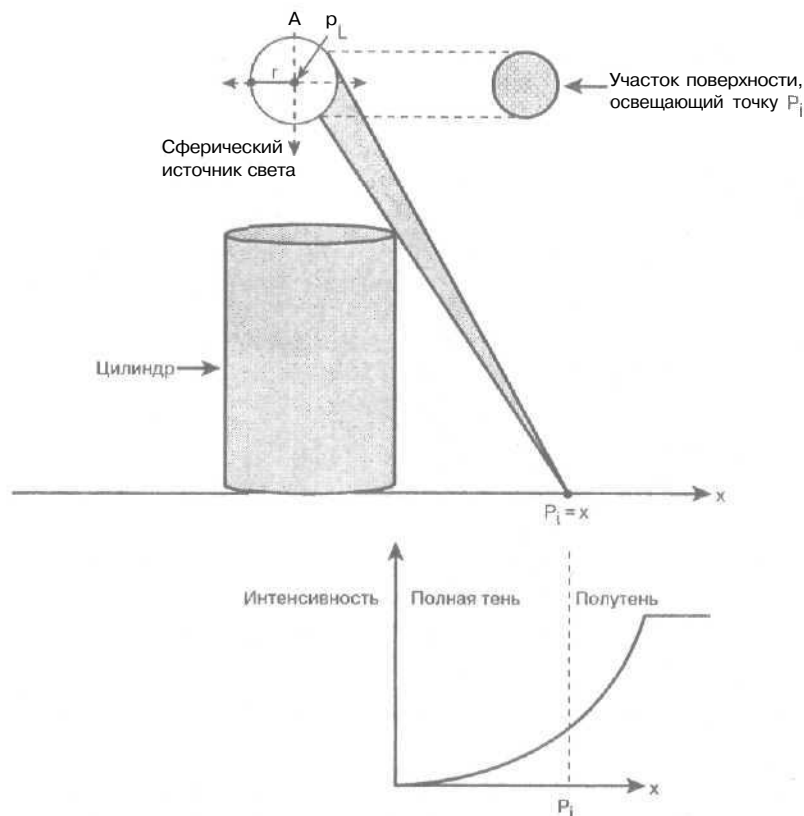


Рис. 14.3. Трассировка лучей сферического источника света

Эти точки определяют область сферы, которая создает полное освещение целевой точки, и по мере перемещения последней по оси  $x$  ее освещенность увеличивается или уменьшается. Этот эффект и порождает полутень: при перемещении контрольной точки от границ тени интенсивность света возрастает (тень уменьшается), поскольку поверхность области, освещающей эту позицию, увеличивается (на рис. 14.46 показана визуализированная сцена с полутенью). При сохранении позиции точки и увеличении размера источника, освещающего эту точку, тень становится более размытой (рис. 14.4а и 14.4б).

Главный вывод, который мы должны сделать из вышесказанного, состоит в том, что для корректного моделирования полной тени и полутени следует учитывать размер источника освещения и выполнять определенную трассировку лучей. Но это может быть сопряжено с большим количеством вычислений. Очевидно, что их вполне можно выполнить, но вопрос в том, действительно ли это необходимо? Даже сейчас, когда я пишу эту книгу, во многих играх тени вовсе не используются, в других используются резкие тени, без полутеней. Однако в некоторых играх используются мягкие тени, и эти игры выглядят великолепно (обычно полутени реализованы с помощью множественного текстурирования, пиксельного затенения и буферов шаблонов).

Поскольку в течение почти четверти столетия видеоигры были без теней (или, по крайней мере, без теней, корректных с точки зрения физики), любая тень лучше, чем никакая. Поэтому, даже если не рассчитывать точную интенсивность света в каждом

пикселе, все равно можно создать искусственную мягкую тень при помощи альфа-смешивания или других "хитрых" приемов — этого обычно более чем достаточно.

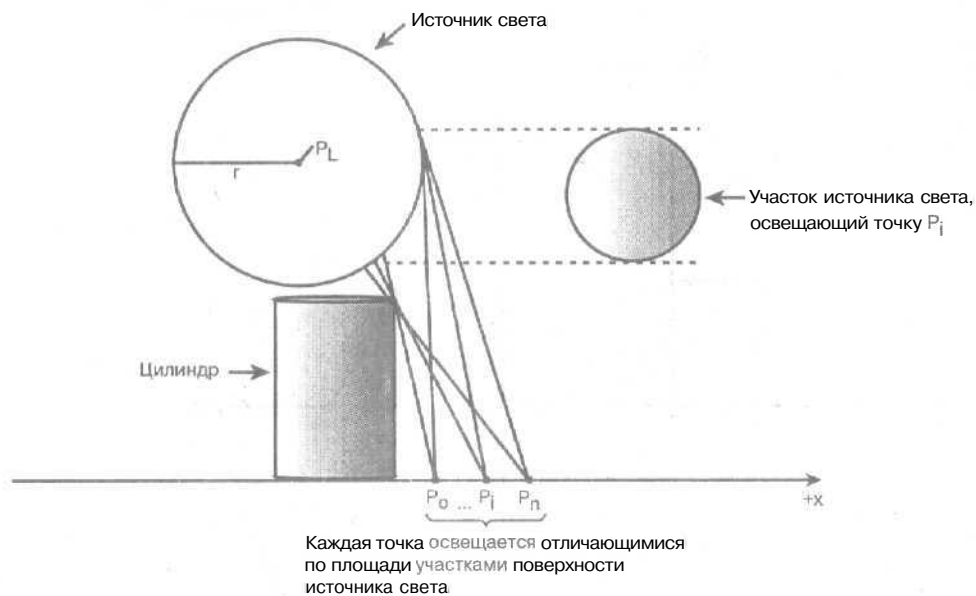


Рис. 14.4а. Увеличение размера источника освещения создает более размытые тени

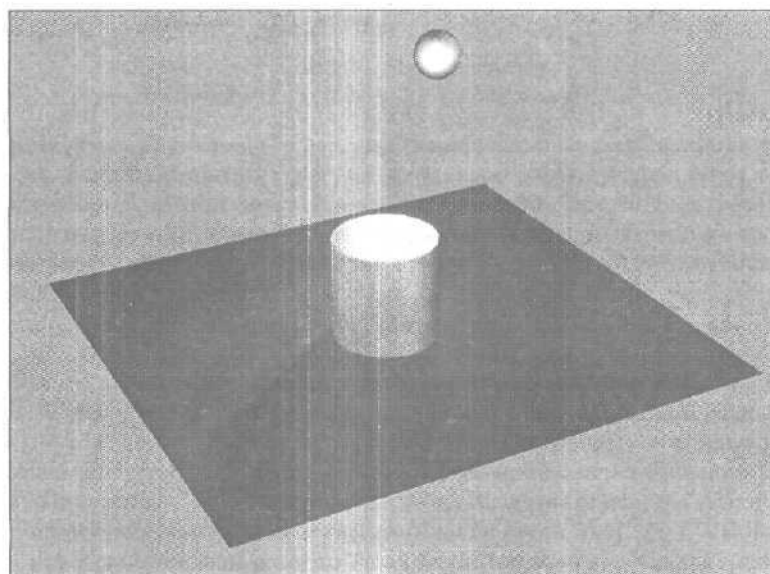


Рис. 14.4б. Размытость тени из-за размера источника освещения

Теперь, когда вам известно, что порождает тени с оптической точки зрения, и как образуется полная тень и полутень, можно перейти к пошаговому созданию теней — от очень простых до весьма реалистичных.

## Имитация теней с помощью проецирования изображений и макетов

Первый способ создания теней вряд ли имеет отношение к реальной тени, соответствующей геометрии объекта, который отбрасывает тень. Но тем не менее, тень выглядит очень убедительно — до тех пор, пока вы не посмотрите на нее внимательнее. Согласно используемой мною технологии, изображение тени просто рисуется под объектом, который, как предполагается, порождает ее. Например, на рис. 14.5 представлен хороший пример проективной тени персонажа. Здесь **тень** — это не что иное, как круглое пятно, расположенное под персонажем.

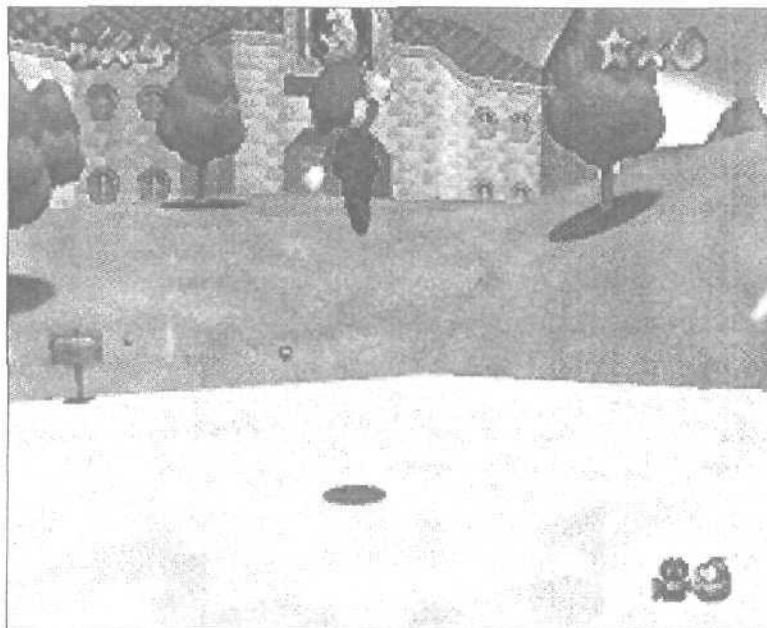


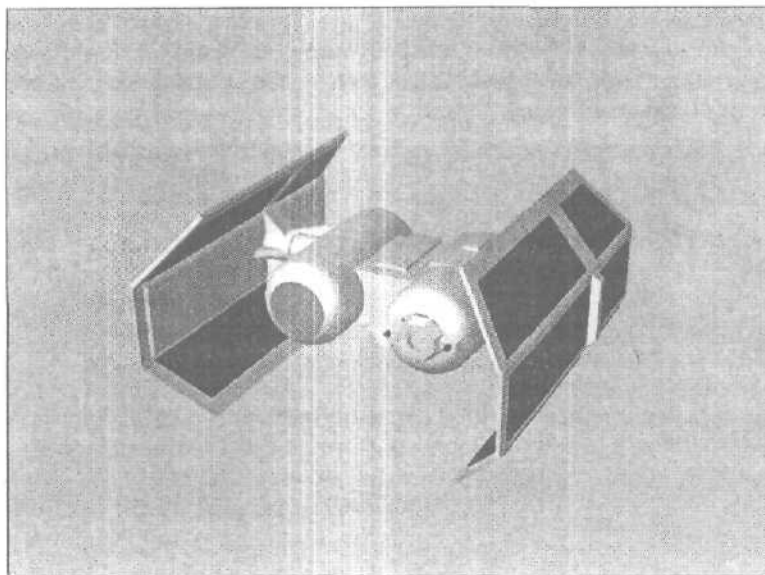
Рис. 14.5. Пример двумерной тени, визуализируемой при помощи растрового изображения

В самом простом случае тень находится точно под персонажем, и в ее положении не учитывается реальное наличие какого-либо источника света. Следующим уровнем детализации должно быть масштабирование размера тени по мере передвижения персонажа вверх и вниз, благодаря чему тень будет становиться больше или меньше. Это также довольно обычное явление.

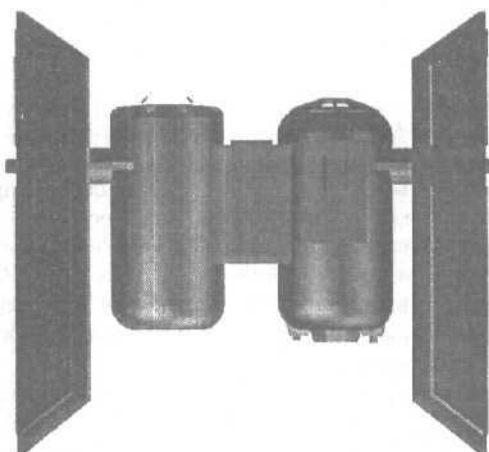
Кроме **того**, во многих играх в качестве тени используется проекция изображения объекта, а не круга. Например, взгляните на рис. 14.6а. На нем изображен трехмерный космический истребитель, а на рис. 14.6б показано изображение тени, отбрасываемой объектом сверху вниз (разумеется, размеры образа тени могут быть изменены). Этот битовый образ можно использовать в тех случаях, когда хочется немного большей детализации.

Однако в этом случае могут возникнуть проблемы. Ведь каждому объекту будет **необходим** свой битовый образ тени. Это не так существенно, однако вид тени зависит от ориентации объекта в пространстве, и, например, если объект вращается, то и тень тоже должна вращаться. Это не создает особых проблем, если битовый образ используется

в качестве текстуры **многоугольного** макета, как показано на рис. 14.7 — в этом случае можно **вращать** сетку, что приведет к вращению текстуры. Однако для первой попытки создания теней лучше все же создать один образ "тени" для всех объектов. Потом **можно** будет попытаться использовать свой образ тени для каждого объекта, должным образом отслеживая ориентацию объекта в пространстве при его **вращении**, и в соответствии с этим вращать или трансформировать его тень.



*Рис. 14.6а. Космический истребитель*



*Рис. 14.6б. Двумерная проекция образа истребителя, используемая в качестве текстуры тени*

Далее, при **следующей** детализации, которую можно реализовать в модели тени, необходимо не только изменять масштаб тени в соответствии с высотой объекта, **создающего**

тень, но для реалистичности использовать в расчетах один из источников **освещения**, чтобы определить, куда Именно отбрасывается тень (рис. 14.8). Для определения местоположения тени мы собираемся использовать только центр объекта, **создающего** тень, и центр источника света. Даже в такой упрощенной ситуации тень выглядит весьма неплохо. **Единственное**, что мы опускаем — это искажение тени, поскольку мы всегда рисуем один и тот же битовый образ (возможно, повернутый). Реализация искажения будет рассматриваться в следующем разделе, посвященном плоским теням; пока же мы будем придерживаться этого упрощенного варианта.

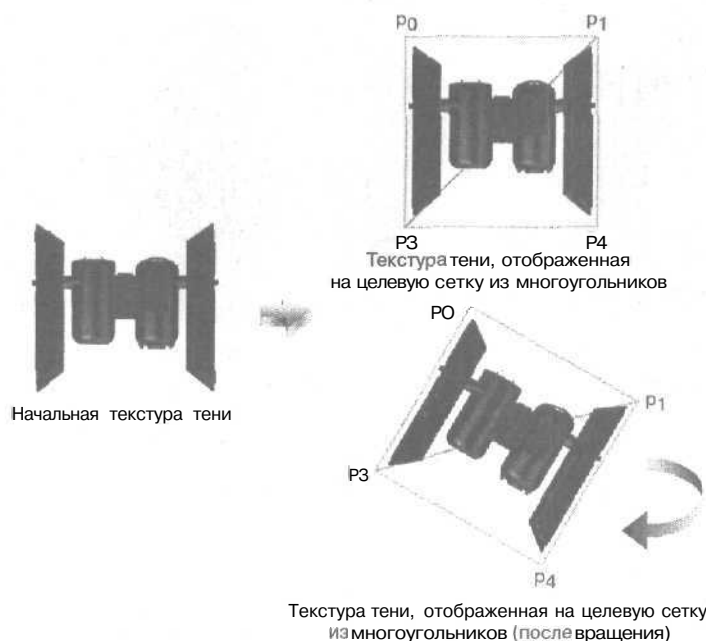


Рис. 14.7. Отображение текстуры тени на сетку для упрощения вращения

Следующее, что мы собираемся **сделать**, — это реализовать **вышеизложенные** идеи. Но прежде необходимо научиться рисовать **макеты**, или плоские прямоугольники с отображенными текстурами, которые обладают определенной прозрачностью (что совершенно необходимо для **эффективной** визуализации тени). Замысел заключается в том, чтобы создать квадрат из двух треугольников, а затем отобразить в него в качестве тени текстуру черной кругообразной области. Но в тех местах, где находятся прозрачные пиксели, мы должны **обеспечить** видимость сквозь них. Это в некоторой степени является проблемой, поскольку наши **растеризаторы** не поддерживают прозрачность. Следовательно, для того, чтобы добавить такую поддержку тени, нам необходимо еще раз переписать некоторые из **растеризаторов**.

## Растеризация с поддержкой прозрачности

Почти все, что необходимо для рисования теней или, в общем случае, для вывода макетов, у нас уже есть. Мы собираемся создать простую сетку из двух многоугольников, показанную на рис. 14.9, и затем отобразить на нее текстуру тени. В настоящий момент единственной проблемой является то, что все наши функции отображения текстур в состоянии работать только с непрозрачными текстурами.

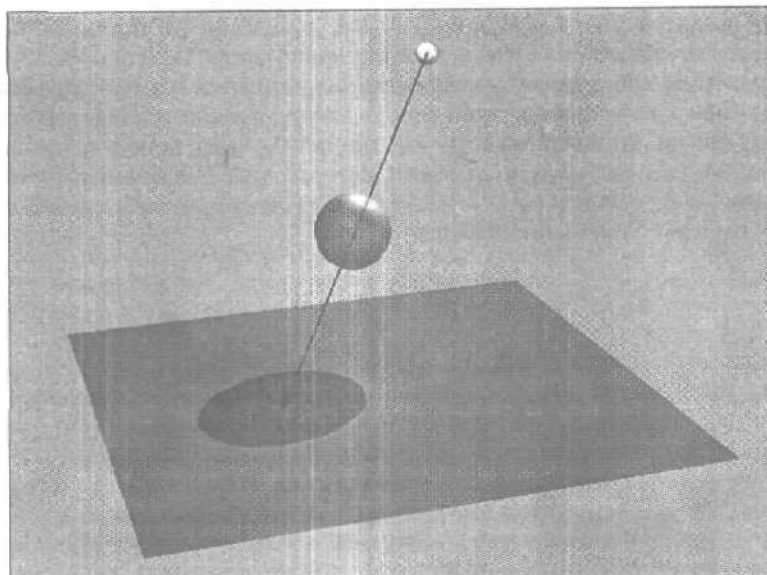


Рис. 14.8. Масштабирование и проецирование тени на горизонтальную плоскость

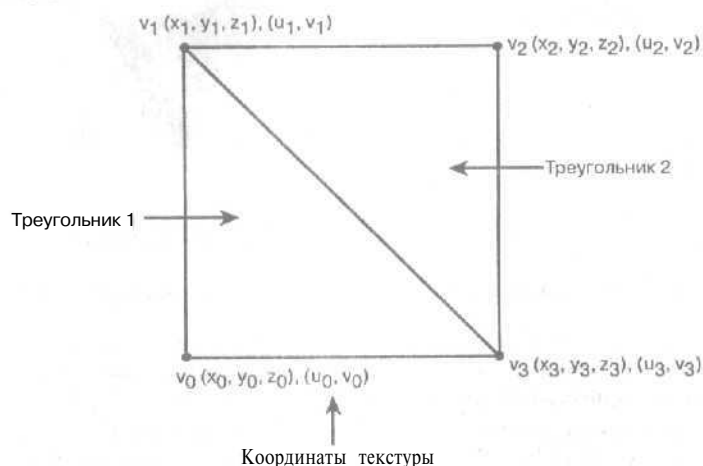


Рис. 14.9. Макет на основе простой сетки из двух треугольников

#### СОВЕТ

Макет (billboard) — это общий термин, используемый в трехмерном программировании для обозначения многоугольника с отображенным на него двумерным изображением в качестве текстуры. Макет обычно сохраняет свою пространственную ориентацию параллельно плоскости просмотра или некоторой другой плоскости. Макеты используются для реалистичного вывода двумерных проекций трехмерных объектов на двумерную плоскость. Во многих трехмерных играх можно увидеть макеты, представляющие деревья, здания или другие объекты, которым не нужна полная трехмерная полигональная сетка (огонь и взрывы являются хорошими примерами макетов; они почти никогда не делаются из многоугольников — это всего лишь двумерные изображения, которые в пространственном отношении всегда ориентированы в направлении к вам). В нашем случае мы собираемся реализовать макеты только для вывода теней.

Добавление поддержки прозрачности к растеризаторам, с которыми мы работали до настоящего времени, довольно простая задача. Тем не менее, я хочу напомнить, что прозрачность — это не альфа-смешивание. Прозрачность в этом контексте означает *полную* прозрачность, т.е. если пиксель имеет значение  $i_{rgb}$ , то он не визуализируется и не растеризуется. Таким образом, если имеется текстура, содержащая в себе отверстия со значениями пикселей, равными в области отверстий  $i_{rgb}$ , то сквозь них можно увидеть фон. Нарис 14.10 это проиллюстрировано с помощью ограды, изготовленной из прутьев, сквозь которую виден визуализированный ландшафт.

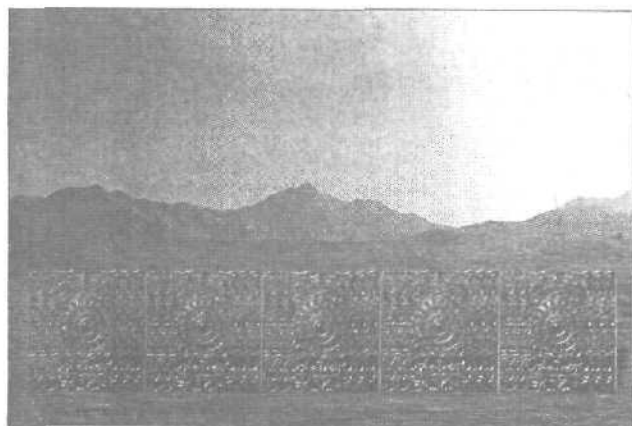


Рис. 14.10. Пример отображения прозрачной текстуры

Поскольку я являюсь приверженцем скорости, я предпочитаю не использовать некоторое обобщенное значение для прозрачного цвета. Предлагаю использовать конкретное значение — `RGB(0,0,0)`; при этом значение типа `SHORT` 16-битового цвета в текстуре будет равно 0, и мы можем добавить единственное сравнение во внутренние циклы нашего кода.

```
if (pixel)
{
    // Визуализация пикселя
} // end if
```

Или выполнить следующее, что дает тот же результат.

```
if (!pixel)
{
    // не выводить!
} // end if
```

В любом случае, 0 — наиболее удобное число для выполнения проверок; при этом имеется множество способов их оптимизации. Единственным недостатком использования значения `RGB(0,0,0)` в качестве прозрачного цвета является то, что он одновременно является черным цветом. Что делать, если в текстуре есть черный цвет, который нужен нам непрозрачным? Воспользуйтесь значениями `(0,0,1)` или `(1,1,1)`, и я гарантирую, что вы не увидите разницы.

Учитывая все сказанное, я не собираюсь переделывать каждый отдельный растеризатор, но все же предоставляю вам возможность использования прозрачных текстур, по крайней мере, для визуализации с Z-буферизацией. Если нужна поддержка 1/z-буферизации, то вы сами

сможете ее добавить (для обновления одного **растеризатора** потребуется около пяти минут работы). Так или иначе, прежде чем показать все прототипы новых функций, я приведу основной цикл из **растеризатора** с отображением аффинной текстуры с постоянным затенением и поддержкой прозрачности со значением **RGB(0,0,0)**.

```
// Вывод участка
for (xi=xstart; xi < xend; xi++)
{
    // Запись с использованием z-буфера

    // Проверка пикселя на прозрачность
    texel = textmap[(ui >> FIXP16_SHIFT) +
        ((vi >> FIXP16_SHIFT) << texture_shift2)];
    if (texel)
    {
        // Модификация z-буфера
        z_ptr[xi] = zi;
        // Запись пикселя
        screen_ptr[xi] = texel;
    } // end if

    // Интерполяция u,v,z
    ui+=du;
    vi+=dv;
    zi+=dz;
} // end for xi
```

Я выделил строку кода, которая совершенно необходима для поддержки прозрачности. Перед добавлением поддержки прозрачности код выглядел следующим образом.

```
// Вывод участка
for (xi=xstart; xi < xend; xi++)
{
    // Запись с использованием z-буфера

    // Запись пикселя
    screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
        ((vi >> FIXP16_SHIFT) << texture_shift2)];
    // Модификация z-буфера
    z_ptr[xi] = zi;

    // Интерполяция u,v,z
    ui+=du;
    vi+=dv;
    zi+=dz;
} // endforxi
```

Если сравнить старый код с новым, **включающим** поддержку прозрачности, то можно увидеть, что новый код просто считывает элемент текстуры **texel** и сравнивает его со значением **RGB(0,0,0)**, и при отличии от указанного значения продолжает процесс визуализации вместе с модификацией **Z-буфера**.

**СОВЕТ**

Очень важно то, что **Z-буфер** не модифицируется при наличии прозрачных пикселей, Этих пикселей, по сути, **нет**, так что они не могут влиять на **Z-буфер**.

## Новый библиотечный модуль

Новый код и растеризаторы находятся в файлах T3DLIB12.CPP\H. Далее приводится заголовочный файл.

```
// T3DLIB12.H - файл заголовка для T3DLIB12.H

#ifndef T3DLIB12
#define T3DLIB12

// Внешние переменные //////////////////////////////////////
extern HWND main_window_handle; // Дескриптор окна
extern HINSTANCE main_instance; // Экземпляр приложения

// Прототипы //////////////////////////////////////

// Новая функция контекста визуализации, которая вызывает
// растеризаторы с поддержкой прозрачности

void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_3(
    RENDERCONTEXTV1_PTR rc);

// Обычная z-буферизация с записью и поддержкой прозрачности
// текстур
void Draw_Textured_TriangleGSWTZB2_16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Textured_TriangleFSWTZB3_16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

void Draw_Textured_TriangleWTZB3_16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера

// Z-буферизация и поддержка прозрачности в текстурах
void Draw_Textured_TriangleZB3_16(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобuffer
    int mem_pitch,      // Байтов в строке
    UCHAR *_zbuffer,    // Указатель на Z-буфер
    int zpitch);        // Байтов в строке Z-буфера
```

```
// Обычная z-буферизация с записью и поддержка прозрачности
// и альфа-смешивания
void Draw_Textured_TriangleWTZB_Alpha16_2(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch,          // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_TriangleZB_Alpha16_2(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке
    UCHAR *_zbuffer,     // Указатель на Z-буфер
    int zpitch,          // Байтов в строке Z-буфера
    int alpha);

void Draw_Textured_Triangle_Alpha16_2(
    POLYF4DV2_PTR face, // Указатель на поверхность
    UCHAR *_dest_buffer, // Указатель на видеобуфер
    int mem_pitch,       // Байтов в строке
    int alpha);
#endif
```

Из заголовочного файла видно, что есть шесть новых **растеризаторов**, которые идентичны предыдущим версиям (на основе которых они построены), но с добавлением поддержки прозрачности. Я переделал достаточное количество функций, чтобы быть уверенным в том, что мы сможем рисовать текстуры на основе альфа-смешивания с поддержкой прозрачности как с помощью Z-буферизации, так и без нее. Однако, если помните, мы перестали вызывать растеризаторы вручную и используем функцию "контекста визуализации". Ранее мы использовали версию 2.0; теперь же следует использовать версию 3.0 функции.

```
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_3(
    RENDERCONTEXTV1_PTR rc);
```

Внешне между функциями нет других различий, кроме наличия в имени цифры 3. Разумеется, в этой функции предусмотрена поддержка растеризаторов с **прозрачностью**. При использовании прозрачности имеется возможность небольшого ускорения. Я мог бы добавить атрибуты многоугольника флажок для указания прозрачности текстуры, а затем использовать инструкцию switch для определения того, какой из растеризаторов необходимо использовать, и осуществляющий соответствующее ветвление, но это ускорение не стоит таких хлопот. Кроме того, можно было бы добавить какие-нибудь параметры для указания прозрачности в присоединенной текстуре, но у нас и так имеется достаточно большое количество причудливых параметров настройки для освещения и альфа-смешивания.

Это все, что относится к программному коду. Остальное — это чистая математика, и каждую рассматриваемую технологию мы реализуем в контексте демонстрационных программ.

## Простые тени

Реализация простых теней включает в себя не что иное, как рисование битового образа тени с помощью альфа-смешивания **непосредственно** под объектом, который, как предполагается, отбрасывает тень. Для реализации этого эффекта прежде всего необходимы две вещи:

текстура тени и сетка многоугольников для макета. Начнем с текстуры тени. Имя ее файла — `shadow64_64.bmp`; это изображение размером 64х64 пикселей, представляющее собой очень темный диск `RGB(16,16,16)`. На рис. 14.11 показан негатив текстуры.

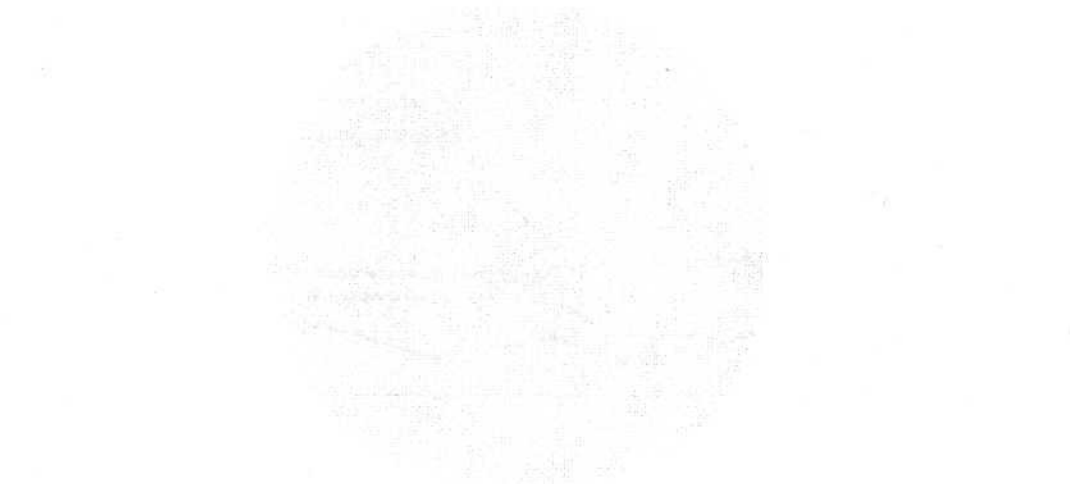


Рис. 14.11. Текстура тени (негативное изображение)

Следующий шаг — моделирование квадрата, построенного из двух треугольников, а затем привязка к сетке текстуры тени. Кроме того, чтобы непрозрачные части текстуры выглядели более реалистичными в качестве теней, в модели необходимо разрешить альфа-смешивание. Если вы помните, поддержка всех требуемых возможностей предусмотрена в формате файла `Caligari .COB`. Мы просто устанавливаем прозрачность для режима фильтрации, а затем — уровень прозрачности. В данном случае я использовал уровень, при котором текстура тени выглядит темной, но не слишком черной. Наконец, модель освещения должна быть такой, чтобы модель макета тени имела постоянное затенение — ведь мы не хотим, чтобы свет отражался от тени!

Окончательная модель макета тени находится в файле `shadow_poly_01.cob` на прилагаемом компакт-диске. Можно было бы показать копию экрана с этим изображением, но показывать по сути нечего, кроме двух треугольников! Так или иначе, но теперь, когда имеется сетка, состоящая из многоугольников, а для многоугольников есть прозрачная текстура с альфа-смешиванием с непрозрачными текстурами, мы готовы к созданию демонстрационного примера.

Написать демонстрационную программу достаточно просто. При создании тени для любого объекта необходимо выполнить следующие действия.

1. Визуализировать все объекты игрового мира с помощью обычной **Z-буферизации**.
2. Выполнить второй проход, и для каждого объекта с тенью разместить сетку тени непосредственно под объектом поверх ландшафта (если таковой ниже объекта) при помощи альфа-смешивания и отображения прозрачных текстур.

Необходимость второго прохода обусловлена тем, что при альфа-смешивании тени с ландшафтом или геометрическими фигурами, находящимися под этой тенью, в буфере необходимо наличие изображения, к которому применяется альфа-смешивание. Кроме того, поскольку предполагается, что тень находится на земле, мы должны выяснить не только то, где находится земля, но и на какой высоте от земли должна находиться тень (рис. 14.12).

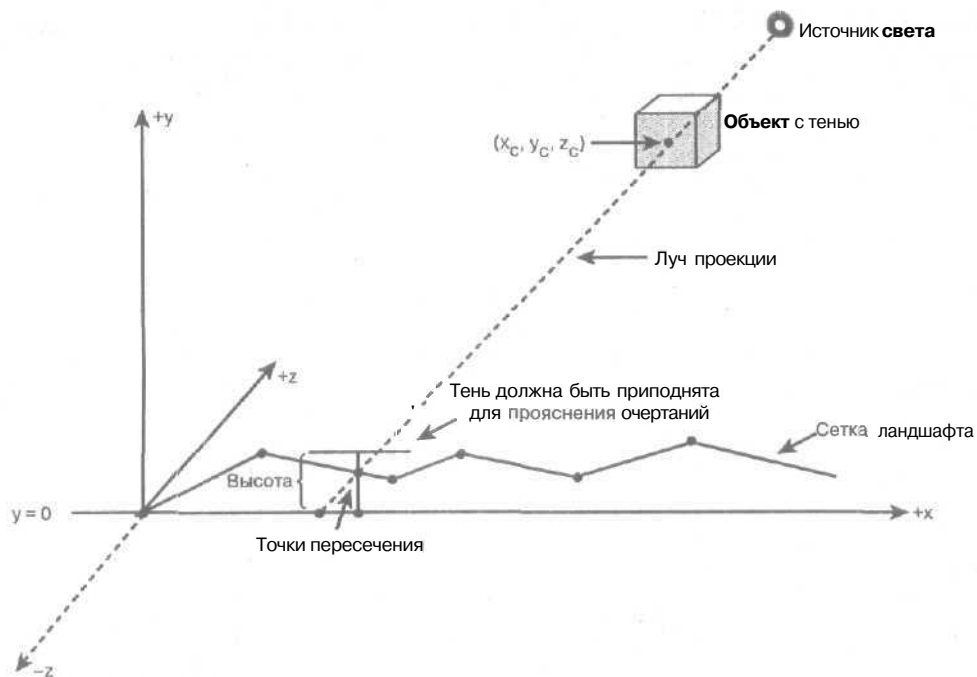


Рис.. 14.12. Вычисление высоты визуализации тени

Для демонстрационного примера, я использовал наш испытанный генератор ландшафта для создания ледяной планеты с небольшой луной, **вращающейся** вокруг ее **центральной** части. Процесс визуализации происходит следующим образом. Ландшафт и планета визуализируются как **обычно**, а затем выполняется второй проход, но на этот раз визуализируется единственный объект — сетка тени, которая располагается непосредственно под объектом и находится на высоте ландшафта. Таким образом, тень не работает с Z-буфером. Разумеется, в случае пересеченной местности тень будет выглядеть недостаточно хорошо из-за того, что она плоская, а ее геометрия не зависит от ваших действий!

Ниже приводится фрагмент кода из демонстрационного примера, иллюстрирующий проход тени.

```
// Сброс списка визуализации
Reset_RENDERLIST4DV2(&rend_list);

// Сброс объекта (имеет значение только для удаления
// объектов и обратных поверхностей)
Reset_OBJECT4DV2(&shadow_obj);

// Вычисление ячейки ландшафта для тени
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) /
obj_terrain.fvar1;
cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) /
obj_terrain.fvar1;

// Вычисление индексов в списке вершин для текущего квадрата
int vO = cell_x + cell_y * obj_terrain.ivar2;
int v1 = vO + 1;
```

```

int v2 = v1 + obj_terrain.ivar2;
int v3 = v0 + obj_terrain.ivar2;

// Поиск высоты по индексам в таблице
terrain_height = MAX(MAX(obj_terrain.vlist_trans[v0].y,
    obj_terrain.vlist_trans[v1].y),
    MAX(obj_terrain.vlist_trans[v2].y,
    obj_terrain.vlist_trans[v3].y));

// Обновление местоположения
shadow_obj.world_pos = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height + 10;

// Создание единичной матрицы
MAT_IDENTITY_4X4(&mrot);

// Преобразование локальных координат объекта
Transform_OBJECT4DV2(&shadow_obj, Smrot,
    TRANSFORM_LOCAL_TO_TRANS, 1);
// Преобразование в мировые координаты
Model_To_World_OBJECT4DV2(&shadow_obj,
    TRANSFORM_TRANS_ONLY);

// Вставка объекта в список визуализации
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &shadow_obj, 0);

```

Далее в полной версии кода устанавливается контекст и выполняется процесс визуализации. На рис. 14.13 показана копия экрана демонстрационной программы. Выглядит неплохо, но к концу главы все будет еще лучше! Демонстрационная программа называется DEMOII14\_1.CPP|EXE, и для ее компиляции необходимы все обычные библиотечные файлы DirectX, а также модули T3DLIB1-12.CPP|H. Подсказка по управляющим клавишам выводится на экран во время работы программы. Самые интересные возможности программы — перемещение, переключение источников освещения и включение каркасного режима работы.

## Масштабирование теней

Теперь, когда у нас есть нечто, напоминающее тень, можно перейти на следующий уровень. Теперь нам надо масштабировать тень, руководствуясь тремя параметрами: размером объекта, его расположением относительно земли (высота), и местонахождением источника освещения, создающего тень.

При вычислениях размер объекта является постоянной величиной, так что он не должен нас волновать. Позднее при расчетах можно будет использовать масштабирующий коэффициент, но высоту объекта и высоту источника освещения необходимо рассматривать уже сейчас в реальном времени. Мы должны получить математические соотношения, которые учитывают, по крайней мере, высоту объекта и источника освещения над землей (рис. 14.14).

На рис. 14.14 представлена двумерная версия проблемы. Без потери общности можно считать, что объект находится на оси  $y$  на некоторой высоте  $h_o$  и имеет радиус  $r_o$ , а источник освещения — над объектом на высоте  $h_i$ . На рисунке показан ряд источников освещения на разных высотах над объектом. Из рисунка видно, что источник света, находящийся

на высоте  $h_i$ , оставит в тени область от начала координат до радиуса  $r_i$ . В общем случае, когда источник освещения находится на высоте  $h_i$ , радиус связанной с ним тени будет равен  $r_i$ . Возникает вопрос: как связаны между собой величины  $r_0$ ,  $r$ ,  $h_0$  и  $h_i$ ?

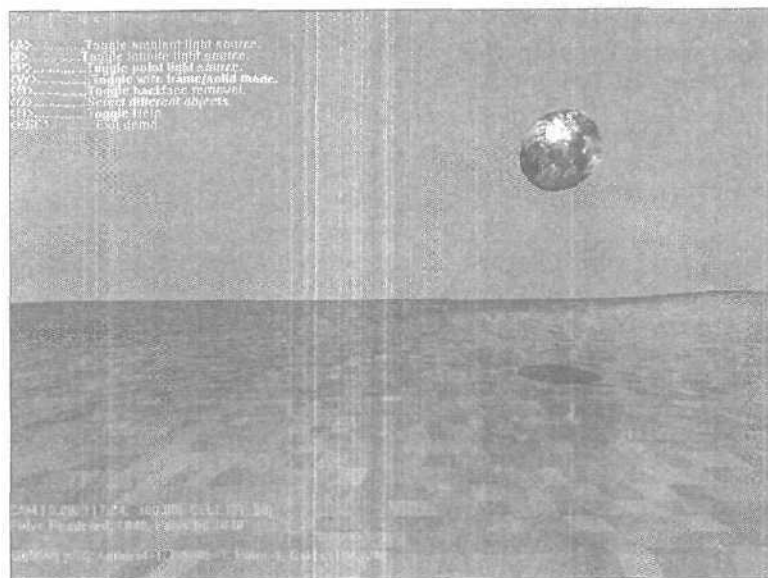


Рис. 14.13. Копия экрана демонстрационного примера простой тени

Решение помогает получить подобные треугольники. Заметим, что треугольник, сформированный началом координат и сторонами  $h_i$  и  $r_i$ , является подобным треугольнику, сформированному центром объекта и сторонами  $(h_i - h_0)$  и  $r_0$ . Отсюда можно вывести следующие соотношения<sup>1</sup>:

$$\frac{r_0}{r_i} = \frac{h_i - h_0}{h_i}.$$

Преобразуя, находим  $r_i$ :

$$r_i = r_0 \frac{h_i}{h_i - h_0}.$$

<sup>1</sup> Насамом деле, данные треугольники не являются подобными, и точное решение с привлечением тригонометрии дает соотношение  $r_i = \frac{r_0 h_i}{\sqrt{(h_i - h_0)^2 + r_0^2}}$ , которое при условии  $L - h_0 \gg r_0$  переходит в полученное автором, которое вполне можно использовать в силу приближенности моделирования тени. — Прим. ред.

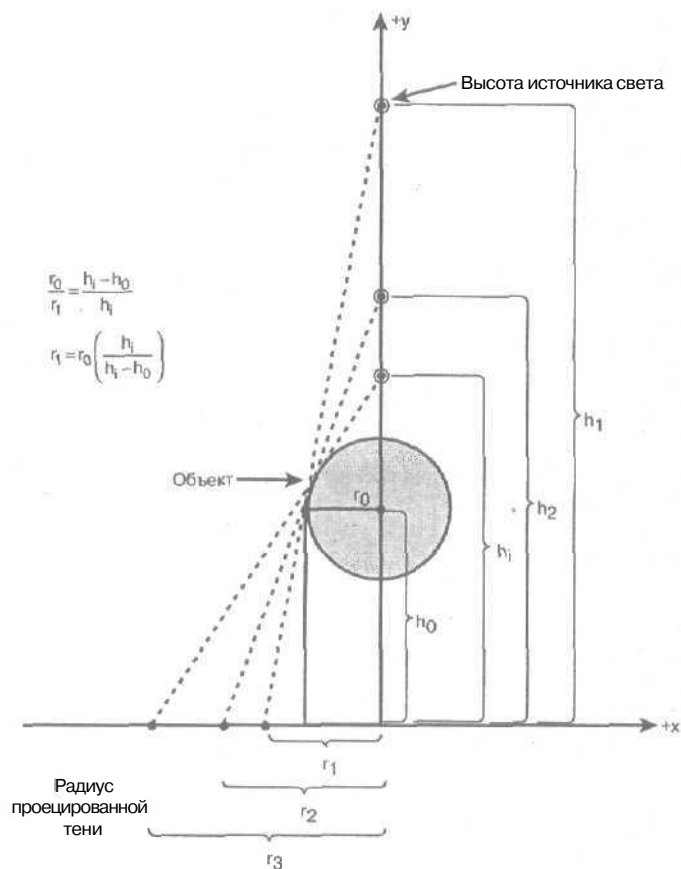


Рис. 14.14. Геометрическая схема масштабирования тени

Давайте проверим это на практике. Что, если источник освещения находится бесконечно далеко? Тогда следует предположить, что световые лучи будут параллельными, и поэтому тень в пределе должна иметь радиус  $r_0$ . Давайте убедимся в этом:

$$\lim_{h_1 \rightarrow \infty} k = \lim_{h_1 \rightarrow \infty} r_1 \frac{h_1}{h_1 - h_0} =$$

$$= r_0 \lim_{h_1 \rightarrow \infty} \frac{1}{1 - \frac{h_0}{h_1}} = r_0.$$

Итак, с увеличением высоты источника освещения ( $h_1 \rightarrow \infty$ ) радиус тени стремится к радиусу объекта. Теперь реализуем полученные результаты в коде.

В следующей демонстрационной программе выполняется масштабирование тени с учетом координат одного из точечных источников освещения.

В наших вычислениях будет использоваться только высота источника освещения (которым будет зеленый точечный источник). Процесс визуализации аналогичен предыдущему, изменяется только окончательный ее размер, для чего сетка тени перед визуализацией масштабируется. Ниже приведен фрагмент кода, который решает эту задачу.

```

// Второй проход визуализации и вывод тени

// Сброс списка визуализации
Reset_RENDERLIST4DV2(&rend_list);

////////////////////////////////////////
// Объект тени

// Сброс объекта (имеет значение только для удаления
// объектов и обратных поверхностей)
Reset_OBJECT4DV2(&shadow_obj);

// Вычисление ячейки ландшафта для тени
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) /
obj_terrain.fvar1;
cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) /
obj_terrain.fvar1;

// Вычисление индексов в списке вершин для текущего квадрата
int v0 = cell_x + cell_y*obj_terrain.ivar2;
int v1 = v0 + 1;
int v2 = v1 + obj_terrain.ivar2;
int v3 = v0 + obj_terrain.ivar2;

// Поиск высоты по индексам в таблице
terrain_height = MAX(MAX(obj_terrain.vlist_trans[v0].y,
obj_terrain.vlist_trans[v1].y),
MAX(obj_terrain.vlist_trans[v2].y,
obj_terrain.vlist_trans[v3].y));

// Обновление местоположения
shadow_obj.world_pos = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height+25;

// Вычисляем радиус тени при помощи выведенных ранее формул,
// в предположении, что тень отбрасывается сферическим
// объектом, освещаемым точечным источником света на высоте
// hl над уровнем земли. Объект имеет вид сферы с радиусом
// ro, расположенной на высоте ho над уровнем земли. Радиус
// тени равен rs:
//  $rs = ro * (hl) / (hl - ho)$ ;

// Высота зеленого источника света
hl = lights2[POINT_LIGHT_INDEX].pos.y;

// Используем средний радиус объекта для вычислений - это
// все же лучше, чем ничего
float rs = ks * (obj_work->avg_radius[0] *
(hl / (hl - obj_work->world_pos.y)));

// Генерируем масштабирующую матрицу
MAT_IDENTITY_4X4(&mrot);

// Устанавливаем коэффициенты масштабирования
mrot.M00 = rs;
mrot.M11 = 1.0;
mrot.M22 = rs;

```

```
// Выполняем масштабирование локальных координат объекта при
// помощи матрицы
Transform _OBJECT4DV2(&shadow_obj, &mrot,
    TRANSFORM_LOCAL_TO_TRANS, 1);

// Преобразование в мировые координаты
Model_To_World_OBJECT4DV2(&shadow_obj,
    TRANSFORM_TRANS_ONLY);

// Внесение объекта в список визуализации
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &shadow_obj, 0);
```

СОВЕТ

Обратите внимание на константу *ks*, используемую при вычислении масштаба. Это поправочный множитель, который применяется для получения подходящего размера тени. Иногда правильная модель выглядит менее реально, чем неточная...

Я выделил полужирным шрифтом добавленный код, который по сути соответствует проведенным математическим расчетам. Единственно интересным является тот факт, что для масштабирования сетки я решил использовать матрицу преобразования. Если помните, то при масштабировании вершины *p* при помощи матрицы ***M<sub>x</sub>***, последняя имеет следующий вид:

$$M_x = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Разумеется, в нашем случае мы используем только масштабирование на плоскости *xz*.

На рис. 14.15 показана копия экрана демонстрационной программы **DEMOIII14\_2.CPP|EXE**. Для ее компиляции в дополнение к библиотечным файлам **DirectX** необходимы файлы **T3DLIB1-12.CPP|H**. Управляющие клавиши данной программы идентичны клавишам предыдущей демонстрационной программы, с добавлением управления высотой зеленого точечного источника света с помощью клавиш **<1>** и **<2>**. Попробуйте изменить высоту источника света и посмотрите, что будет происходить с тенью. Но не поднимайте источник света слишком высоко — иначе иллюзия будет разрушена! Удостоверьтесь также при помощи переключения в каркасный режим с помощью клавиши **<W>**, что сетка, представляющая тень — это обычный квадрат.

## Отслеживание источника света

Создаваемая нами тень становится все более реалистичной. Пока что она всегда имеет вид диска — в качестве объекта, отбрасывающего тень, можно использовать хоть микроавтобус, но тень все равно останется диском. Но привести ее в соответствие с объектом достаточно просто. Реальной же проблемой является то, что тень всегда находится непосредственно под объектом, а это неправильно. В предыдущей демонстрационной программе мы использовали зеленый точечный источник освещения, но при этом нас интересовала только его высота, необходимая для вычисления радиуса проецируемой тени. Теперь пора научиться определять положение тени на плоскости *xz* с учетом положений как объекта, так и источника света.

Для получения данных, необходимых для решения этой задачи, обратимся к рис. 14.16. Я полагаю, что для разнообразия в решении можно использовать параметри-

**ческие** векторные уравнения, а не подобные треугольники. На рисунке показаны положение точечного источника  $\vec{p}_1$ , центр объекта  $\vec{p}_0$ , а соответствующая точка проекции на плоскость земли —  $\vec{p}_s$ .

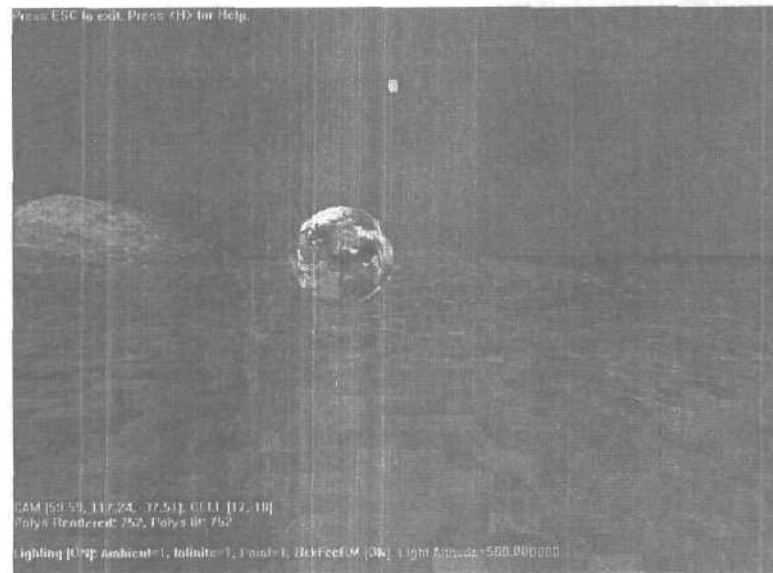


Рис. 14.15. Копия экрана демонстрационной программы с поддержкой масштабирования тени

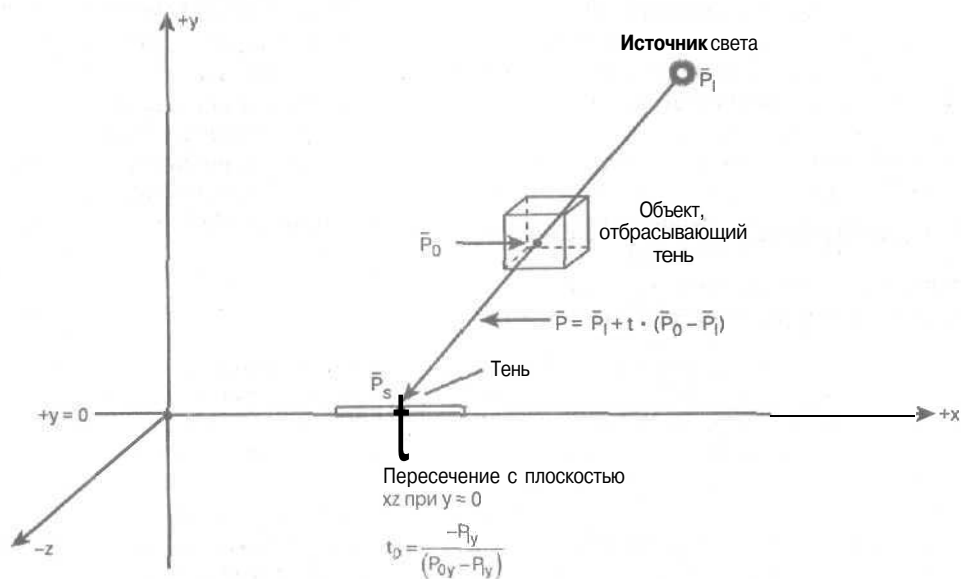


Рис. 14.16. Схема вычисления положения тени с учетом **положения** источника света

Запишем **интересующие** нас уравнения и **посмотрим**, что мы можем определить с их помощью. Параметрическая прямая в трехмерном пространстве, проходящая через ис-

точник освещения и центр объекта, записывается как

$$\mathbf{p}_s = \mathbf{p}_l + (\mathbf{p}_0 - \mathbf{p}_l)t.$$

Распишем это уравнение покомпонентно:

$$p_{sx} = p_{lx} + (p_{0x} - p_{lx})t,$$

$$p_{sy} = p_{ly} + (p_{0y} - p_{ly})t,$$

$$p_{sz} = p_{lz} + (p_{0z} - p_{lz})t.$$

Нас интересуют координаты  $x$  и  $z$  точки  $\mathbf{p}_s$ , когда  $p_{sy} = 0$ . Таким образом, нам достаточно подставить  $p_{sy} = 0$  во второе уравнение, найти  $t$ , а затем для полученного значения  $t$  из первого и третьего уравнений найти координаты  $x$  и  $z$  тени. Найти  $t$  очень просто:

$$t = -p_{ly} / (p_{0y} - p_{ly}).$$

Имея точку  $\mathbf{p}_s$ , мы выполним процесс визуализации теневого объекта с центром в этой точке. Кроме того, для завершения системы отображения тени можно добавить эффект масштабирования.

Реализация конвейера визуализации осталась неизменной; она такая же, как и в предыдущем демонстрационном примере. Выполняется процесс визуализации ландшафта и объекта, затем определяется местоположение теней с помощью только что рассмотренных математических расчетов, и при втором проходе визуализации выполняется процесс альфа-смешивания теневых объектов (с прозрачными текстурами) на горизонтальной плоскости ландшафта.

Разумеется, форма тени все еще не имеет никакого отношения к объекту, поскольку мы всегда используем дискообразную тень; кроме того, нет никакого отсечения или деформации тени. Но позже мы добьемся и этого.

Учитывая сказанное, рассмотрим новый программный код реализации тени, где местоположение тени вычисляется с учетом положения как объекта, так и источника света.

```
// Сброс списка визуализации
Reset_RENDERLIST4DV2(&rend_list);

// Сброс изображения тени для автомобиля
cockpit.curr_frame=0;

int v0, v1, v2, v3; // Используются для отслеживания вершин

VECTOR4D pl,      // Положение источника света
po,              // Положение объекта с тенью
vlo,             // Вектор от источника света к объекту
ps;              // Положение тени
float rs,         // Радиус тени
t;               // Параметр t
////////////////////
// Объект тени

// Сброс объекта (имеет значение только для удаления
// объектов и обратных поверхностей)
Reset_OBJECT4DV2(&shadow_obj);

// Вычисление ячейки ландшафта для тени
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) /
obj_terrain.fvar1;
```

```

cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) /
obj_terrain.fvar1;

// Вычисление индексов в списке вершин для текущего квадрата
v0 = cell_x + cell_y*obj_terrain.ivar2;
v1 = v0 + 1;
v2 = v1 + obj_terrain.ivar2;
v3 = v0 + obj_terrain.ivar2;

// Поиск высоты по индексам в таблице
terrain_height = MAX(MAX(obj_terrain.vlist_trans[v0].y,
obj_terrain.vlist_trans[v1].y),
MAX(obj_terrain.vlist_trans[v2].y,
obj_terrain.vlist_trans[v3].y));

// Обновление местоположения
// shadow_obj.world_pos = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height+25;

// Используем точечный источник света 1 в качестве
// прожектора, вычисляем положение проекции (используя
// локальные переменные для облегчения понимания
// математических вычислений)

// Определяем положение точечного источника света
pl = lights2[POINT_LIGHT_INDEX].pos;

// Определяем положение объекта
po = obj_work->world_pos;

// Создаем вектор от источника света к объекту
VECTOR4D_Build(&pl, &po, &vlo);

// Теперь самое интересное— найти t когда координата y
// прямой проекции равна 0. Технически мы размещаем тень
// немного выше, так что если вам хочется это учесть, то вы
// можете изменить приведенную формулу для t соответствующим
// образом
t = -pl.y / vlo.y;

// Теперь вычислим координаты x,z проецируемой тени
// (масштабирование значения t выполняется для более
// близкого размещения тени)
shadow_obj.world_pos.x = pl.x + t*vlo.x;
shadow_obj.world_pos.z = pl.z + t*vlo.z;

// Вычисляем величину радиуса тени
// rs = ro * (hl)/(hl - ho)

// Высота источника освещения
hl = lights2[POINT_LIGHT_INDEX].pos.y;

// Используем среднее значение радиуса объекта
rs = ks * ( obj_work->avg_radius[0] *

```

```

(hl/(hl - obj_work->world_pos.y));

// Генерируем масштабирующую матрицу для тени
MAT_IDENTITY_4X4(&mrot);

// Устанавливаем коэффициенты масштабирования
// для плоскости xz
mrot.M00 = rs;
mrot.M11 = 1.0;
mrot.M22 = rs;

// Масштабируем локальные координаты объекта
Transform_OBJECT4DV2(&shadow_obj, &mrot,
    TRANSFORM_LOCAL_TO_TRANS,1);

// Преобразование локальных координат в мировые
Model_To_World_OBJECT4DV2(&shadow_obj,
    TRANSFORM_TRANS_ONLY);

// Вставка объекта в список визуализации
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list,
    &shadow_obj,0);

////////////////////////////////////

// Проверка, не находится ли камера в тени
VECTOR4D vd;
VECTOR4D_Build (&cam.pos, &shadow_obj.world_pos, &vd);
float d = VECTOR4D_Length_Fast (&vd);

// Проверка на расположение в пределах полутора радиусов
// тени, что делает эффект заметнее
if (d < 1.5*rs)
    cockpit.curr_frame - 1; // Уменьшение яркости кабины
else
    cockpit.curr_frame = 0; // Увеличение яркости кабины

```

Код практически дословно осуществляет параметрическую проекцию и пересечение; единственной особенностью является добавление кода для изменения изображения кабины, когда камера находится в виртуальной тени. При входе в область тени **освещение** кабины уменьшается. Я использовал визуализацию маленькой кабины джипа, и у меня есть два ее варианта: освещенный и темный. Когда выясняется, что камера находится в тени, я изменяю яркое изображение на темное и наоборот, создавая таким образом иллюзию того, что автомобиль находится в тени.

В реальной игре, в которой вы были бы главным действующим лицом, вы, вероятно, использовали бы реальную трехмерную геометрию для кабины, игрока, пистолета или для чего-нибудь еще, а для реалистичности предоставили бы модели распространения света управлять формированием теней. Однако модели распространения света ничего не известно о тенях, и поэтому вы должны сами определять, находится ли сетка в тени, после чего соответственно уменьшать окружающее освещение или делать что-то иное, чтобы заставить сетку стать более темной. Выше я выделил полужирным шрифтом тот фрагмент кода, который выполняет эту небольшую проверку и смену битового образа.

Еще одним дополнительным кодом в последнем демонстрационном примере является код, который выполняет процесс визуализации тени для каждого отдельного точечного источника **освещения** (имеется два источника), и это — ключевой момент при выполнении вычислений теней. Ведь процесс визуализации тени должен быть выполнен для каждого **объекта** и для всех источников освещения. Даже при использовании простейших алгоритмов визуализации тени ситуация может быстро выйти из-под контроля. Представьте себе 50 объектов с тремя источниками **света** — итого 150 дополнительных процессов визуализации! Конечно, каждый из них представлен сеткой только с двумя треугольниками с текстурой, но альфа-смешивание резко понижает производительность, так что тут есть над чем задуматься.

Я хочу убедиться, что вы хорошо поняли весь изложенный к этому моменту материал. Давайте с этой целью рассмотрим блок-схему, представленную на рис. 14.17, которая иллюстрирует конвейер визуализации тени.

1. Визуализация ландшафта.
2. Визуализация световых маркеров.
3. Визуализация объекта с тенью.
4. Растеризация изображения.
5. Визуализация теней.
6. Растеризация изображения с помощью альфа-смешивания.



Рис. 14.17. Блок-схема процесса визуализации тени

На рис. 14.18 представлена копия экрана демонстрационной программы DEMOII14\_3.CPP|EXE. Управление здесь такое же, как и в предыдущем примере, но в этот раз можно управлять высотой и зеленого и красного точечного источника света с помощью клавиш <1> и <2> (для зеленого источника) и клавиш <3> и <4> (для красного). Попробуйте изменять их высоту и наблюдайте за изменением картины. Например, если поместить источники света слишком низко, то это может привести к инвертированию проекции, и тени окажутся совершенно не там, где надо — попробуйте! Прделайте также эксперимент, поместив точечные источники очень высоко над горизонтальной плоскостью для имитации солнечного света. Вы заметите, что в этом случае размеры теней оказываются одинаковыми, а их положение — под объектом, отбрасывающим тень.

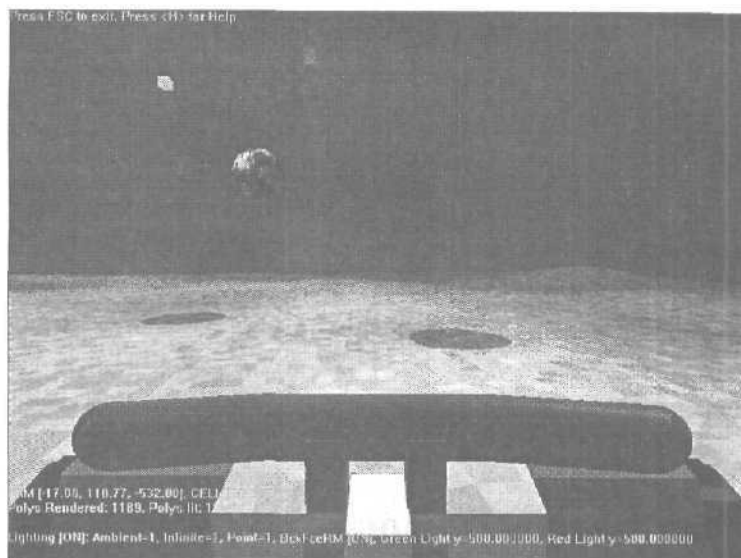


Рис. 14.18. Копия экрана демонстрационной программы с масштабированием и позиционированием тени

Для компиляции демонстрационной программы необходимы библиотечные модули T3DCLB1-12.CPP|H, основная программа DEMOII14\_3.CPP|H и библиотечные файлы DirectX.

## Последние замечания об имитации теней

Даже с тем, что у нас есть на данный момент, можно создавать довольно сложные теневые эффекты. Одна из последних возможностей, которую можно было бы добавить, — это использование битового образа объекта, спроецированного на трехмерную сетку с учетом источника освещения. В этом случае тень выглядит как нечто, напоминающее сам объект. Единственная проблема состоит в том, что при повороте объекта сетка тени также должна поворачиваться. Однако эта задача легко решается, если вращение выполняется параллельно оси *u*, как показано на рис. 14.19.

Если объект вращается вокруг оси *u*, то сетку тени надо вращать точно так же, а в качестве текстуры использовать спроецированное растровое изображение объекта. Конечно, при вращении вокруг другой оси задача существенно усложняется, но для многих игр и объектов можно избежать подобных неприятностей. Добавить описанную возможность достаточно просто (от двух до пяти строк программы), а дополнительный растровый образ действительно сделает вид тени более убедительным.

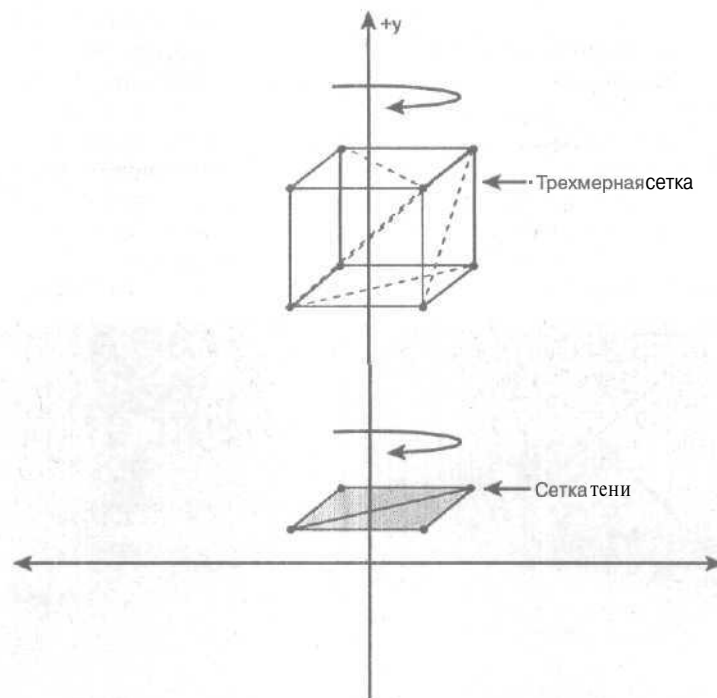


Рис. 14.19. Вращение битового образа сетки тени в соответствии с вращением объекта

## Отображение тени на плоскую сетку

Мы теперь готовы к следующему уровню технологии создания теней, которая использует реальное проецирование сетки на горизонтальную плоскость с последующим выводом с помощью серого цвета и альфа-смешивания так, чтобы она выглядела похожей на тень. По сути, мы моделируем траектории фотонов от источника освещения до вершины на сетке объекта, который отбрасывает тень, и до горизонтальной поверхности земли (рис. 14.20).

По сути, мы уже решили эту проблему в предыдущем примере, теперь нам необходимо только немного обобщить полученное решение. Что нам действительно необходимо сделать, так это найти преобразование, которое "проецирует" трехмерную сетку на плоскую поверхность (в нашем случае на плоскость  $xz$ ), а затем выполнить процесс визуализации полученной плоской сетки с последующим затенением таким образом, чтобы она выглядела похожей на тень (в большей или меньшей степени отключив текстуры и освещение). Другими словами, мы собираемся выполнить процесс визуализации в качестве тени объекта, спроецированного на горизонтальную плоскость.

## Вычисление векторных преобразований для проекции

Итак, нам необходимо преобразование, которое способно спроецировать любую вершину исходной сетки на горизонтальную плоскость с учетом положения источника освещения. Воспользуемся математическими расчетами с немного отличающейся системой обозначений.

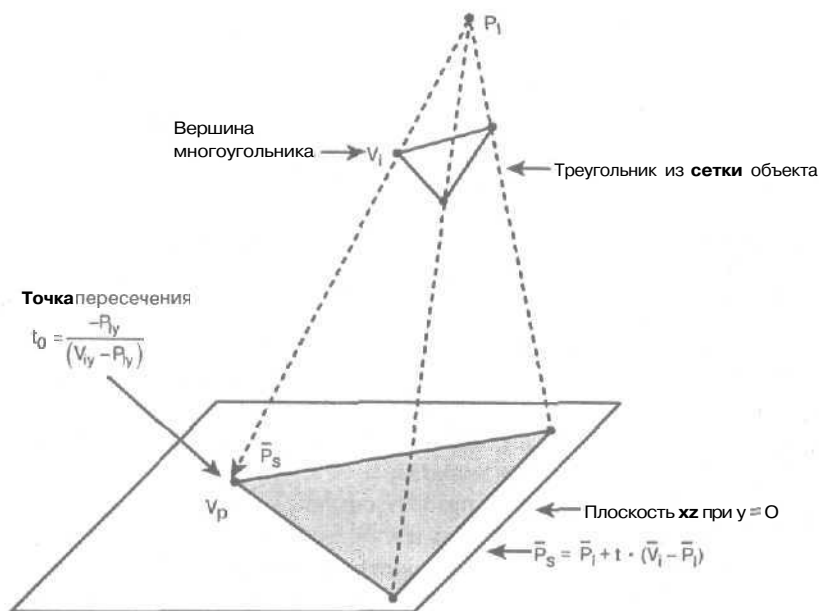


Рис. 14,20, Отображение плоской сетки тени

Мы хотим проецировать вершину  $v_i$  сетки, и эта вершина принадлежит многоугольнику  $p_i$ . Источник освещения, служащий точкой проекции, находится в точке  $p_i$ , а горизонтальная плоскость описывается уравнением  $y = 0$ . Результат проекции вершины на плоскость назовем  $p_s$ ; эта точка вычисляется из уравнения прямой

$$p_s = p_i + (v_i - p_i)t.$$

Запишем это уравнение покомпонентно:

$$p_{sx} = p_{ix} + (v_{ix} - p_{ix})t,$$

$$p_{sy} = p_{iy} + (v_{iy} - p_{iy})t,$$

$$p_{sz} = p_{iz} + (v_{iz} - p_{iz})t.$$

Из второго уравнения, подставляя в него значение  $p_{sy} = 0$ , мы можем найти величину  $t$  (назовем ее  $t_0$ ), а затем подставить найденное решение в остальные уравнения, чтобы найти решения для составляющих  $x, z$ . Итак,

$$t_0 = -p_{iy} / (v_{iy} - p_{iy}),$$

и, соответственно,

$$p_{sx} = p_{ix} + (-p_{iy} / (v_{iy} - p_{iy})) \cdot (v_{ix} - p_{ix}),$$

$$p_{sy} = 0,$$

$$p_{sz} = p_{iz} + (-p_{iy} / (v_{iy} - p_{iy})) \cdot (v_{iz} - p_{iz}).$$

То же самое можно записать при помощи векторов:

$$\begin{aligned} p_s &= p_i + (v_i - p_i)t_0 = \\ &= v_i t_0 + p_i (1 - t_0). \end{aligned}$$

Эта запись идеально подходит для использования при вычислениях с помощью матриц. Вот как выглядит искомая матрица:

$$M_{ps} = \begin{bmatrix} t_0 & 0 & 0 & 0 \\ 0 & t_0 & 0 & 0 \\ 0 & 0 & t_0 & 0 \\ p_{ix}(1-t_0) & p_{iy}(1-t_0) & p_{iz}(1-t_0) & 1 \end{bmatrix}$$

При умножении вершины на данную матрицу мы получим **следующий** результат (убедитесь в его корректности сами):

$$v_i \cdot M_{ps} = [v_{ix}t_0 + p_{ix}(1-t_0) \quad v_{iy}t_0 + p_{iy}(1-t_0) \quad v_{iz}t_0 + p_{iz}(1-t_0)]$$

Здесь, однако есть одна **небольшая** проблема. Попробуйте-ка сами разобраться, какая. Догадались? Эта матрица годится только для одной конкретной вершины, поскольку значение  $t_0$  вычисляется с использованием  $v_i \dots$  Поэтому, хотя матричная операция и вполне **допустима**, но она допустима только для конкретной вершины  $v_i$ , и для другой вершины матрицу надо пересчитывать заново.

Так что использование матричного преобразования — не настолько хорошая идея, как это казалось сначала. Лучше просто решать векторное уравнение вручную для каждой вершины, находя значение  $t_0$  для  $y = 0$ , затем подставляя его и находя  $x$  и  $z$  для каждой вершины. Но нет худа без добра, и поскольку мы делаем все преобразования и вычисления в мировых координатах, мы можем опустить шаг позиционирования сетки **тени**.

Теперь, когда у нас есть плоская сетка тени, мы можем приступить к ее визуализации, но то, что мы получим, не будет выглядеть как тень, пока мы не изменим свойства ее затенения. Ниже описано, какие изменения должны быть внесены для **того**, чтобы тень объекта выглядела тенью.

1. Возьмите сетку объекта, для которого нужно спроецировать тень, преобразуйте ее, спроецируйте локальные координаты на плоскость и сохраните их в переменных для преобразованных координат. (Запомните, никакие действия с сеткой не могут повредить ее до тех пор, пока локальные координаты модели будут в порядке. Их всегда можно преобразовать и скопировать в список преобразованных вершин.)
2. Теперь, когда у нас есть в наличии плоская сетка, измените ее освещение и затенение таким образом, чтобы сетка стала похожей на тень. Просто сохраните цвет и атрибуты каждого многоугольника, а затем перепишите поверх затенение с постоянной интенсивностью, альфа-смешение с величиной, равной примерно 50%, и отключите текстуру.
3. Передайте сетку в конвейер визуализации и восстановите ранее сохраненные цвет и атрибуты.

По сути, если объект отбрасывает тень, то мы используем непосредственно сам объект для создания тени, проецируя его сетку на горизонтальную плоскость, а потом изменяем параметры затенения сетки, чтобы она выглядела как реальная тень.

Ниже приводится фрагмент кода из очередной демонстрационной программы, который все это реализует.

```
// Сброс списка визуализации
Reset_RENDERLIST4DV2(&rend_list);
```

```
////////////////////////////////////
// Преобразуем объект в тень, проецируя его вершины на
// горизонтальную плоскость
```

```

// Сброс объекта (имеет значение только для удаления
// объектов и обратных поверхностей)
Reset_OBJECT4DV2(obj_work);

// Сохраняем атрибуты/цвет затенения каждого многоугольника,
// заменяем их атрибутами тени; позже мы восстановим их
int pcolor[OBJECT4DV2_MAX_POLYS], // Сохранение цвета
    pattr[OBJECT4DV2_MAX_POLYS]; // Сохранение атрибутов

// Сохраняем цвета и все атрибуты каждого многоугольника
for(int pindex = 0; pindex < obj_work->num_polys; pindex++)
{
    // Сохраняем атрибуты и цвет
    pattr[pindex] = obj_work->plist[pindex].attr;
    pcolor[pindex] = obj_work->plist[pindex].color;

    // Устанавливаем атрибуты тени
    obj_work->plist[pindex].attr = POLY4DV2_ATTR_RGB16 |
        POLY4DV2_ATTR_SHADE_MODE_CONSTANT |
        POLY4DV2_ATTR_TRANSPARENT;
    obj_work->plist[pindex].color = RGB16Bit(0,0,0) +
        (2 << 24);
} // end for pindex

// Создаем единичную матрицу
MAT_IDENTITY_4X4(&mrot);

// Находим t для момента пересечения прямой проекции
// горизонтальной плоскости
pl = lights2[POINT_LIGHT_INDEX].pos;

// Преобразуем каждую локальную вершину сетки объекта и
// сохраняем результат в списке преобразованных вершин
for(int vertex=0; vertex < obj_work->num_vertices; vertex++)
{
    POINT4D presult; // Результат преобразования

    // Вычислим параметр to при пересечении с плоскостью y=0
    VECTOR4D vi;

    // Преобразуем координаты в мировые
    VECTOR4D_Add(&obj_work->vlist_local[vertex].v,
        &obj_work->world_pos,&vi);
    float to = -pl.y / (vi.y - pl.y);

    // Преобразуем точку
    obj_work->vlist_trans[vertex].v.x =
        pt.x + to*(vi.x - pl.x);
    obj_work->vlist_trans[vertex].v.y =
        25.0; // pl.y + to*(vi.y - pl.y);
    obj_work->vlist_trans[vertex].v.z =
        pl.z + to*(vi.z - pl.z);
}

```

```
obj_work->vlist_trans[vertex].v.w = 1.0;
} // end for index
```

```
// Вставка объекта в список визуализации
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, obj_work, 0);
```

Затем используется следующий источник освещения для создания еще одной сетки, и так далее, пока не будут спроецированы все источники освещения (в нашем случае их только два). Далее восстанавливаются старые атрибуты и цвета, а сетка возвращается к нормальному виду:

```
// восстановим атрибуты и цвет
for (pindex=0; pindex<obj_work->num_polys; pindex++)
{
    // сохраним атрибуты и цвет
    obj_work->plist[pindex].attr = pattr[pindex];
    obj_work->plist[pindex].color = pcolor[pindex];
} // end for pindex
```

Порядок визуализации аналогичен предыдущему: сначала визуализируется ландшафт, затем световые маркеры и сам объект. Далее выполняется второй проход, в котором визуализируются тени, поскольку им необходимо альфа-смешение с фоновым изображением.

Результаты проекции плоской сетки выглядят вполне реалистично. На рис. 14.21а показана копия экрана демонстрационной программы в каркасном режиме, а на рис. 14.21б — в режиме затенения. Теперь мне хочется сделать тени, которые являются действительно законченными трехмерными сетками, только размазанными (да, это такой технический термин).

Рассмотренная нами демонстрационная программа находится в файлах DEMO114\_4.CPP|EXE. Для ее компиляции необходимы библиотечные модули T3DLIB1-12.CPP|H вместе с библиотечными файлами DirectX. Управление программой такое же, как и предыдущей, но я настоятельно советую вам посмотреть на работу программы в каркасном режиме (включается клавишей <W>), чтобы увидеть деформацию сетки, которая приводит к созданию геометрически правильной тени.

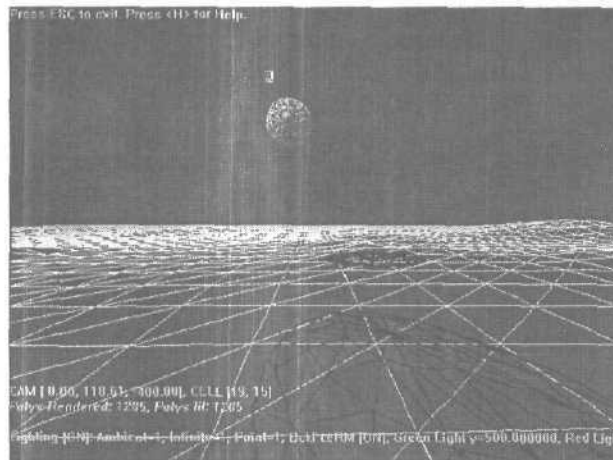


Рис. 14.21а. Копия экрана демонстрационной программы с плоской тенью в каркасном режиме



Рис. 14.216. Копия экрана демонстрационной программы с плоской тенью в режиме сплошного затенения

## Оптимизация плоских теней

Теперь плохие **новости** — мы визуализируем много лишнего. Ведь по **существу** мы сжали законченную объемную модель в двумерную. Таким образом, мы, вполне **вероятно**, визуализируем невидимые или перекрывающиеся поверхности или нечто подобное. На самом же деле необходимо проецировать в качестве теневой сетки только силуэт объекта.

Посмотрите на рис. 14.22, на котором изображен простой куб и обведен его силуэт. Хитрость заключается в поиске силуэта, и для решения этой задачи имеется множество приемов и методов. Силуэт состоит из ребер, разделяющих видимые и невидимые (задние) многоугольники относительно источника освещения.

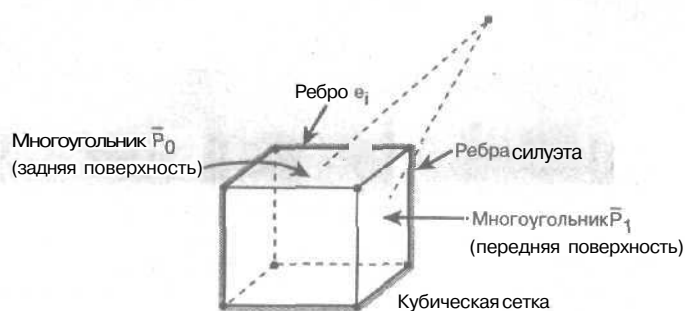


Рис. 14.22. Силуэт куба

Ребро, помеченное на рисунке как  $e_i$ , является частью силуэта. Из рис. 14.22 видно, что это общее ребро для граней  $P_0$  и  $P_1$ . Многоугольник  $P_0$  с точки зрения источника света — невидимая задняя поверхность, тогда как многоугольник  $P_1$  — передняя поверхность. Можно попробовать алгоритмически найти **соответствующие** грани и, соответственно, найти силуэт объекта. Проблема в том, что этот метод будет работать только в случае с объектами без отверстий и самопересечений. Это тоже неплохо, но вопрос состоит в том, стоит ли вообще этим заниматься.

Я полагаю, что использование силуэта вполне подходит для различных алгоритмов, основанных на буферах шаблонов или объемных тенях. Проблема в том, что во многих случаях повышение производительности не стоит времени, которое затрачивается на это повышение. Поэтому другой прием заключается в проецировании упрощенной сетки (*действительно* упрощенной сетки) при формировании тени для каждого объекта. Эта упрощенная сетка может состоять из не более чем десятой части от реального количества многоугольников, которые составляют полную модель, так что дополнительные 10% работы для получения теней — это не так уж и плохо.

## Введение в отображение освещения и кэширование поверхности

Эта тема достаточно трудна, поэтому прежде чем я скажу что-нибудь, я бы хотел, чтобы вы посмотрели на трехмерную сцену без отображения освещения. Посмотрите на рис. 14.23а.

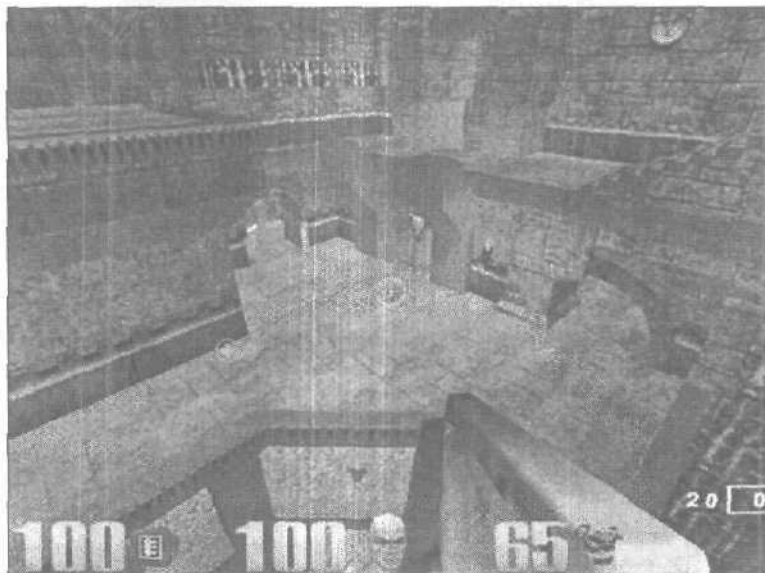


Рис. 14.23а. Уровень игры Quake без отображения освещения

А теперь посмотрите на рис. 14.23б. Видите разницу? Вы можете удивиться, насколько реалистичную картину дает метод затенения по Гуро. Но на самом деле это не так. Во-первых, затенение по методу Гуро основано на вершинах, поэтому с его использованием нельзя получить ничего особо детализированного. Во-вторых, у нас просто нет метода создания теней на стенах, объектах, самозатенения и т.п. Все это достигается при помощи процесса отображения освещения (light mapping).

Методика отображения освещения *весьма* стара и представляет собой по сути рисование текстур в текстурах. Вместо выполнения освещения в реальном времени все вычисления, *касающиеся* освещения, выполняются до начала игры. Кроме того, художник может сделать это вручную и нарисовать сцену так, что она будет выглядеть как освещенная. Полученный результат вычислений, или двумерный рисунок, называется *картой освещения*.

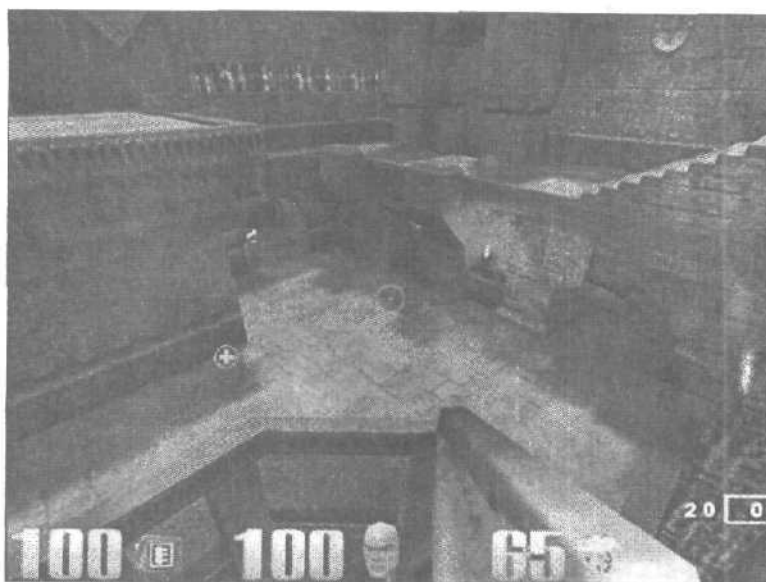


Рис. 14.236. Уровень игры Quake с отображением освещения

Например, если мы возьмем текстуру малого размера (рис. 14.24а) и умножим ее на карту освещения, показанную на рис. 14.24б, то в результате получим текстуру на рис. 14.24в. Этот процесс немного напоминает то, что мы уже делали намного раньше при рассмотрении модели освещения. Просто теперь мы собираемся изменить масштаб применения метода до целых помещений и сеток.

Итак, идея отображения освещения заключается в использовании карты освещения для модулирования текстур, налагаемых на геометрию миров и объектов наших игр. Освещение — это только мираж, оно всего лишь нарисовано...



Рис. 14.24. Отображения освещения на отдельную текстуру

Конечно, это производит впечатление, так почему бы действительно не использовать предварительно освещенные текстуры? Это возможно, но что будет в случае большого пространства с 1000 стен и поверхностей, каждая с текстурой размером 256x256, вроде представленной на рис. 14.25? Потребуется 1000 ее копий, причем все они должны быть **предосвещенными**, а это повлечет за собой существенное повышение требуемого объема памяти. С другой стороны, можно сказать, что в случае использования одной и той же текстуры нам все равно нужны 1000 карт освещения. Это, **конечно**, так — но все же не совсем так. Ключевым моментом является то, что карты распределения освещения **необязательно** должны иметь то же разрешение, что и текстуры. Например, можно **исполь-**

зовать карты освещения из 32x32 элементов для текстур с разрешением 256x256, и это будет практически незаметно.

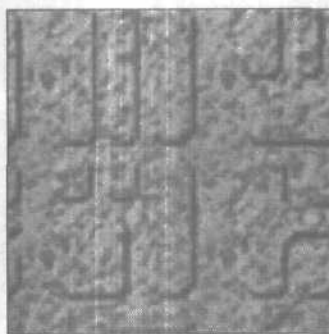


Рис. 14.25. Возможная текстура стены



Рис. 14.26. Использование карт освещенности с низким разрешением

На рис. 14.26 показан пример использования карты освещения с низким разрешением для модулирования текстуры. Тут показана первоначальная текстура, карта **освещения** размером 32x32 и модулированная текстура.

Несомненно, это выглядит немного грубовато, но если пропустить карту освещения через фильтр, то можно получить практически совершенный образ. Итак, мы сохраняем карты освещения с **разрешающей способностью**, во много раз меньшей, чем у целевых текстур (например 32x32), затем разворачиваем их в буфере и усредняем или размываем. Далее мы берем полученные карты нормального размера (те же 256x256) и используем их для модулирования текстур с тем же размером. Получается, что каждая карта **освещения** занимает только  $32 \times 32 = 1024$  байта, а не 65536. **Экономия** — в 64 раза. Результаты при этом выглядят благодаря декомпрессии и усреднению ничуть не хуже.

Конечно, если вы внимательны и **проницательны**, вы увидите, что декомпрессия на самом деле не что иное, как интерполяция карты освещения. Дело в том, что свет по своей природе — это нечто расплывчатое и имеющее неясные очертания, и мягкие тени выглядят лучше теней с резкими краями. Использование карт распределения освещения с низким разрешением и усредняющего фильтра для уменьшения неровностей карты освещения перед модуляцией можно рассматривать в качестве оригинального решения.

Весь процесс для отображения освещения показан на рис. 14.27. Вводится карта **освещения** с низким разрешением, декомпрессируется, затем копируется в буфер с высоким разрешением (разумеется, этот шаг является необязательным). Далее карта освещения используется для модулирования текстуры. Это говорит о том, что значения карты освещения умножаются на значения карты текстуры, а результаты помещаются в буфер и используются в качестве конечной текстуры для отображения на многоугольник (или многоугольники).

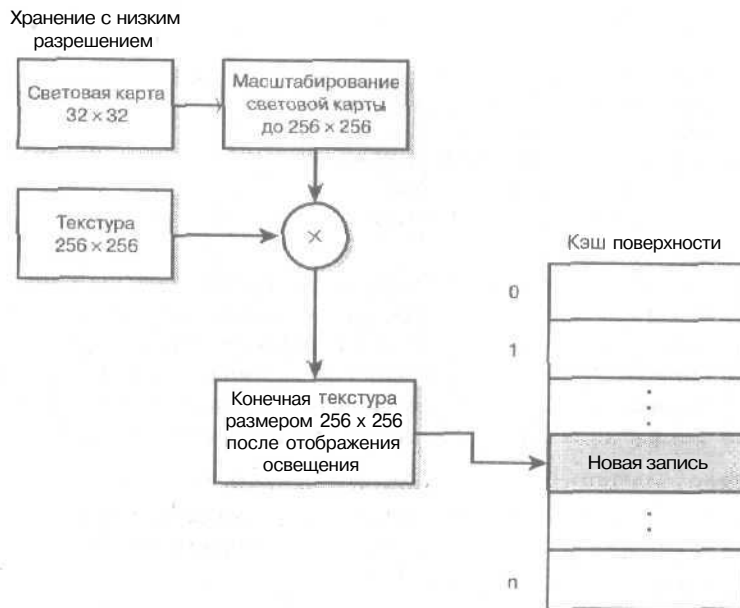


Рис. 14.27. Конвейер отображения освещения

Вот и все, что реально используется для отображения освещения. Возьмем текстуру двумерной карты распределения освещения, используем ее для **модулирования** текстуры в двумерном пространстве, а затем применим полученные результаты для **текстурирования** геометрического объекта. К сожалению, вычислительные затраты при этом могут быть весьма **значительными**, поскольку для текстур размером  $256 \times 256$  необходимо не менее 64000 вычислений, каждое из которых включает от 1 до 3 операций умножения (в зависимости от использования монохромной или RGB-палитры) наряду с операциями сложения и возможными операциями сравнениями. Пусть для вычисления пикселя требуется время  $c_m$ . Тогда для текстуры размером  $m \times n$  и карты освещения того же размера должно быть выполнено, как минимум,  $c_m \cdot t$  -  $n$  вычислений. Это большой объем работ, а главное — всякий раз приходится отображать освещение на текстуру и выполнять ее визуализацию, а затем просто терять все проделанные вычисления! Вот почему необходимо использовать кэширование поверхности.

## Кэширование поверхности

**Кэширование поверхности** — не что иное, как запоминание результатов предыдущих вычислений отображения освещения. Например, **предположим**, что есть 10 текстур, которые используются для отображения освещения в помещении. Вы остаетесь в одном и том же **помещении** — так почему бы не вычислить все версии текстур с отображением освещения и не поместить их в кэш для использования в следующих кадрах? А если в кэше имеется необходимая текстура с отображением освещения, то в ее повторном вычислении нет необходимости. При таком методе работы проведенные вычисления для **отображения** освещения не отбрасываются и позволяют снизить **общее** количество вычислений. По мере устаревания вычисленных текстур, они становятся более не нужными и удаляются. Описать процесс кэширования поверхности с отображением освещения можно **следующим** образом.

Инициализация кэша текстур и удаление **все текстур**

Начало сцены

Для каждой текстуры **T** в сцене

если **T** имеется в кэше поверхности, **визуализировать**  
многоугольник(и) с помощью **T**

иначе вычислить карту освещения для **T**, сохранить  
текстуру в кэше. Если кэш заполнен, удалить  
дольше всего не использовавшиеся текстуры

Конец сцены

Все в **сущности** очень просто. Мы помещаем в кэш результаты вычисления каждого отдельного отображения освещения до тех пор, пока кэш не **заполнится**, и используем результаты из кэша всякий раз, когда выдается запрос, результат для которого имеется в кэше. Так что вам часто удастся совершенно бесплатно получить отображение света и тени в игровом помещении. Это не похоже на правду, но это действительно так!

Разумеется, как **всегда**, есть и неприятные вопросы вроде следующих.

- Насколько большими будут карты распределения освещения?
- Будет ли карта распределения освещения базироваться на многоугольниках или на поверхности (**компланарном** наборе многоугольников) **целиком**?
- Будут ли карты **освещения** сжатыми?
- Будут ли карты **освещения** монохромными или в них будет использоваться RGB-палитра?
- Насколько большим будет кэш поверхности?

Ну и так далее. Ответы на эти вопросы может подсказать только опыт и знание конкретной ситуации. Сейчас же наша цель — просто реализовать **данную** технологию, чтобы показать, насколько она проста и удивительна.

## Генерация карт освещения

Я уже говорил, что карты освещения можно создавать как алгоритмически, так и вручную, с помощью художника. Весь смысл отображения **освещения** заключается в представлении хорошо выполненных теней, темных щелей и так далее. Наилучшим вариантом для этого является независимая система **освещения**, в которой используются трассировка лучей или Какие-либо иные алгоритмы освещения, например, отображение фотона или моделирование световых потоков для построения и визуализации трехмерных изображений сцен.

Реализация трассировки луча не слишком сложна, но ее результаты не столь хороши, как результаты **метода** моделирования световых потоков с учетом диффузного отражения для построения и визуализации трехмерных изображений, которое по существу представляет собой вычисления потока энергии. На рис. 14.28 показано изображение освещенной сцены, полученное в результате использования этого метода. Как видите, выглядит оно очень реалистично. Однако в нашем случае нам прежде всего необходимо все, что касается вычислений карт освещения. Я не собираюсь рассматривать метод моделирования световых потоков для построения и визуализации трехмерных изображений, поскольку он (пока?) не поддерживается в реальном масштабе времени, зато написано уже немало хороших книг, где представляется этот метод наряду с методами отображения фотонов и трассировки лучей.

Второй вариант заключается в использовании инструментария объемного моделирования (например, **Pov-Ray**, **trueSpace** или **3D Studio Max**) для создания полностью **осве-**

щенной монохромной сцены, которую можно использовать в картах освещения с путем изменения положения камеры.

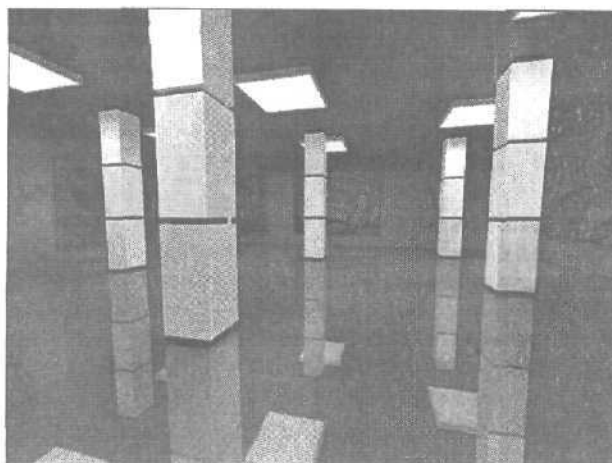


Рис. 14.28. Сцена, полученная с помощью моделирования световых потоков

Наконец, художник может сам "нарисовать" карты освещения и **методом** проб и ошибок создать карты, которые будут выглядеть так же хорошо, или даже лучше, чем **настоящие**. Выбор за вами, но в большинстве случаев, по моему мнению, нужно использовать сочетание этих методов. Например, компания **idSoftware** имеет свои собственные инструментальные средства для вычисления карт освещения. Однако многие компании используют **имеющиеся** инструментальные средства, такие как упоминавшиеся **Pov-Ray** или **3D Studio Max**, и создают только свой интерфейс или средства подключения к **системам**, использующим трассировку лучей или метод моделирования световых потоков, и не тратят времени на написание своих средств. Исходя из своего опыта, я советую вам рисовать свои карты распределения освещения.

## Реализация отображения освещения

А ведь по сути мы уже написали код для отображения **освещения**. Как вы помните, этот алгоритм заключался в модулировании (умножении) одной текстуры с помощью другой. Разумеется, для больших текстур требуется выполнение огромного количества вычислений, но в примере **DEMO118\_4.CPP|EXE** сделано именно это. Я взял код из этого демонстрационного примера, немного его оптимизировал и создал встроенный фрагмент, который использует источник текстуры, карту **освещения** и буфер для модулирования текстуры источника с помощью карты освещения. Результаты получаются очень быстро и великолепно выглядят. Однако такой алгоритм не пригоден для оптимизации в компьютерах с **SIMD-архитектурой** (Pentium III+). Возможно, мы попытаемся выполнить оптимизацию в последней главе книги, но пока ниже приведен неоптимизированный фрагмент кода для отображения освещения на исходную текстуру.

```
// Отображение освещения
```

```
// Идея заключается в том, чтобы взять исходную карту  
// освещения и использовать ее для модулирования текстуры,
```

```
// которая отображается на целевые многоугольники, а затем
// записать результаты в текстуру для визуализации
```

```
// Наш маленький алгоритм обработки изображения :)
// Pixel_dest[x,y]rgb=pixel_source[x,y]rgb*light_map[x,y]rgb
```

```
USHORT *sbuffer=(USHORT *)texture_copy.buffer;
USHORT *lbuffer=(USHORT *)lightmaps[curr_lightmap].buffer;
USHORT *dbuffer=(USHORT *)obj_terrain.texture->buffer;
```

```
// RGB-преобразование битового образа
for (int iy = 0; iy < texture_copy.height; iy++)
    for (int ix = 0; ix < texture_copy.width; ix++)
    {
        int rs,gs,bs;           // Используется для извлечения
                                // исходных rgb значений
        int rl,gl,bl;           // rgb значения карты освещения
        int rf,gf,bf;           // конечные rgb-члены

        // Извлечение пикселя из исходного изображения
        USHORT spixel=sbuffer[iy*texture_copy.width + ix];

        // Извлечение RGB-значения
        _RGB565FROM16BIT(spixel, &rs,&gs,&bs);

        // Извлечение пикселя из карты освещенности
        USHORT lpixel=lbuffer[iy*texture_copy.width + ix];

        // Извлечение RGB-значения
        _RGB565FROM16BIT(lpixel, &rl,&gl,&bl);

        // Вычисление члена модуляции
        rf = ( rs*rl ) >> 5 ;
        gf = ( gs*gl ) >> 6 ;
        bf = ( bs*bl ) >> 5 ;

        // Создание RGB, проверка на переполнение
        // и запись назад в буфер
        dbuffer[iy*texture_copy.width + ix] =
            _RGB16BIT565(rf,gf,bf);
    } // end for ix
```

Алгоритм очень прост: берется исходная текстура из буфера **sbuffer**, которая модулируется с **помощью** карты освещения, находящейся в буфере **lbuffer**, а результаты сохраняются в буфере **dbuffer**. Вычисления выполняются в **RGB-пространстве** для каждого пикселя. Таким образом, есть два цикла — один для *x* и один для *y*, которые выполняют итерации по пространству текстуры, извлекают RGB-значения и для исходной текстуры и для световой карты, перемножают их и запоминают результаты в буфере адресата **dbuffer**. Эти результаты затем используются в качестве конечной текстуры для отображения на сетку многоугольника.

Прежде чем рассмотреть демонстрационную программу отображения **освещения**, давайте посмотрим, что получится без отображения **освещения** (рис. 14.29).

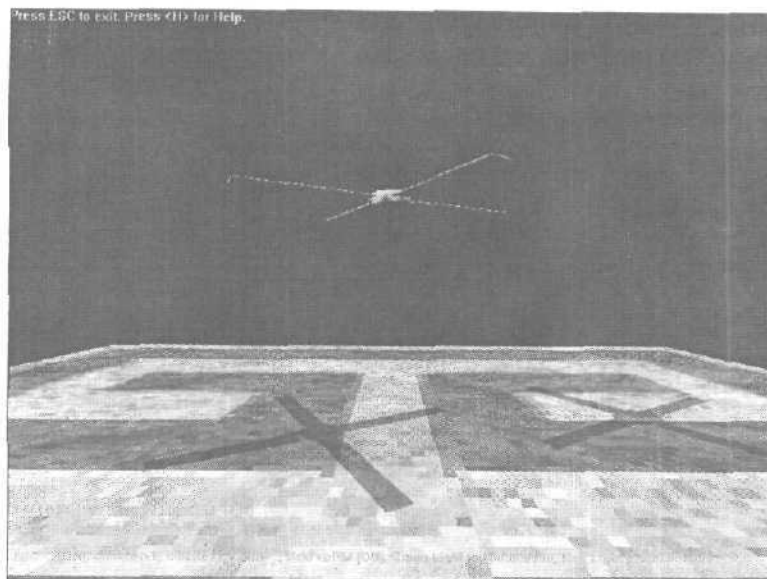


Рис. 14.29. Копия экрана плоской тени от находящегося сверху вентилятора

Здесь вы видите копию экрана демонстрационной программы DEMOII14\_5.CPP|EXE, которая является вводной частью к демонстрационной программе с отображением освещения. По сути, это не что иное, как DEMOII14\_4.CPP|EXE, где в качестве объекта, отбрасывающего тень, используется модель потолочного вентилятора (рис. 14.30).

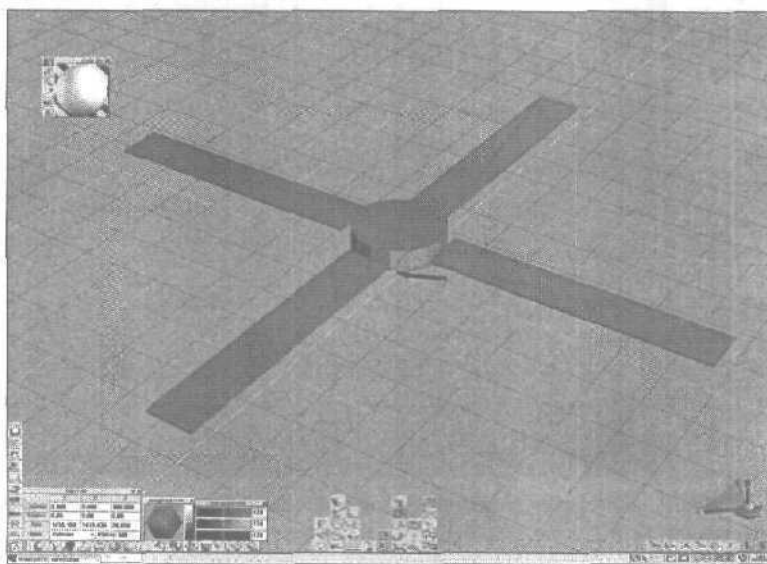


Рис. 14.30. Модель потолочного вентилятора

Теперь давайте посмотрим, как работает пример, демонстрирующий отображение освещения. В первом демонстрационном примере отображения освещения DEMOII14\_6.CPP|EXE

имеется ряд фрагментов для построения демонстрационной программы. Во-первых, мне была необходима модель вентилятора, которую я сделал в trueSpace минут за 5-10. Это просто цилиндр и четыре лопасти. Имя файла с моделью — fan\_01b.cob. Во-вторых, мне потребовалась текстура для пола, которая показана на рис. 14.31.

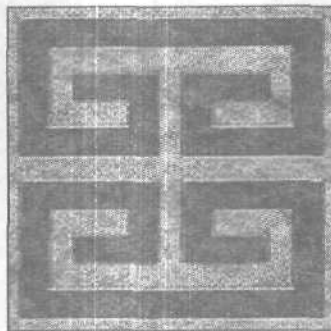


Рис. 14.31. Текстура пола для демонстрационной программы

И наконец, мне необходимы карты освещения. Вообще говоря, это единственная сложная часть демонстрационной программы. Я использовал программу моделирования для создания вида вентилятора сверху, а затем визуализировал их в виде 16-битовых изображений размером 256x256. Установка параметров визуализации показана на рис. 14.32.

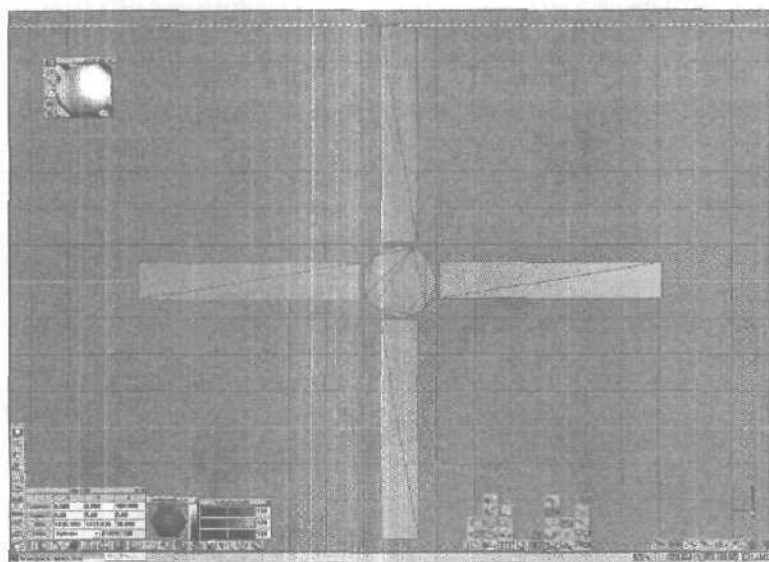


Рис. 14.32. Установка параметров визуализации карты освещения для вентилятора

После выполнения визуализации двумерного битового образа, я поместил битовый образ вентилятора в Paint Shop Pro и поворачивал изображение с приращением в 10°, чтобы создать карты распределения освещения для анимации. Этот процесс показан на рис. 14.33. Теперь, при наличии модели и карт освещения очень просто создать завершенную демонстрационную программу.

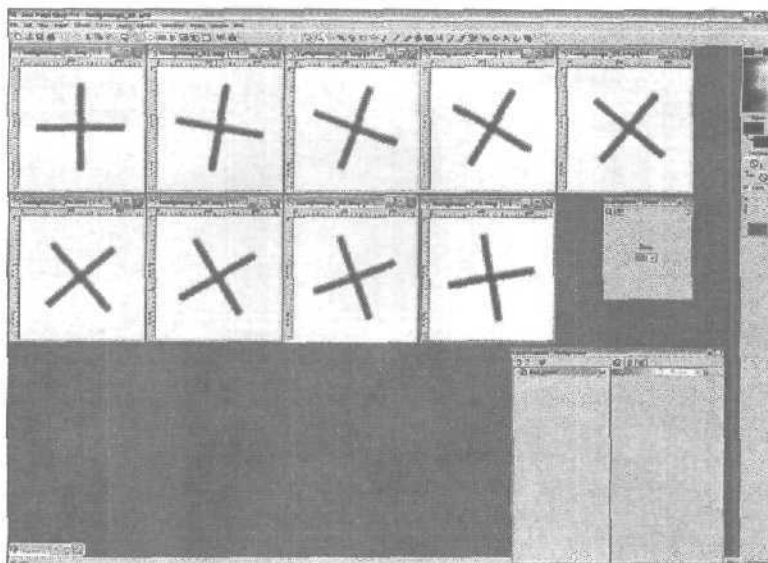


Рис. 14.33. Карты освещения после вращения и обработки изображения

Программа DEMOII14\_16.CPP представляет собой очень простой цикл. Он начинается с получения текущей карты распределения освещения (1 из 9), выбора карты освещения, модулирования текстуры базовой поверхности этой картой и перехода к стадии визуализации. Горизонтальная поверхность визуализируется при помощи текстуры отображения освещения, а затем — за один проход, поскольку в этом случае отсутствует стадия альфа-смешивания — происходит визуализация трехмерной модели верхнего вентилятора. Удивительно, не правда ли? Вся работа проделана предварительно, а на стадии модуляции текстуры выполняется отображение освещения. Результаты весьма впечатляют, и вы сами можете увидеть это на рис. 14.34. Для компиляции демонстрационной программы требуются файлы T3DLIB1-12.CPP|H вместе с главным исходным файлом DEMOII14\_6.CPP, и не забудьте добавить в проект библиотечные файлы DirectX.

## Отображение темноты

Никто не говорил, что мы должны рисовать только тени, — точно так же мы можем нарисовать и освещение. Фактически, термин *отображение освещения* употреблялся неправильно, поскольку на самом деле мы отображали темноту, а не **освещение**. Мы используем алгоритм модуляции, который в лучшем случае не изменяет значение пикселя, а в **большинстве** случаев затемняет его.

В любом случае, я изменил программу DEMOII14\_6.CPP|EXE для загрузки карт освещения с противоположной полярностью, т.е. я взял девять карт и инвертировал их изображения, как показано на рис. 14.35.

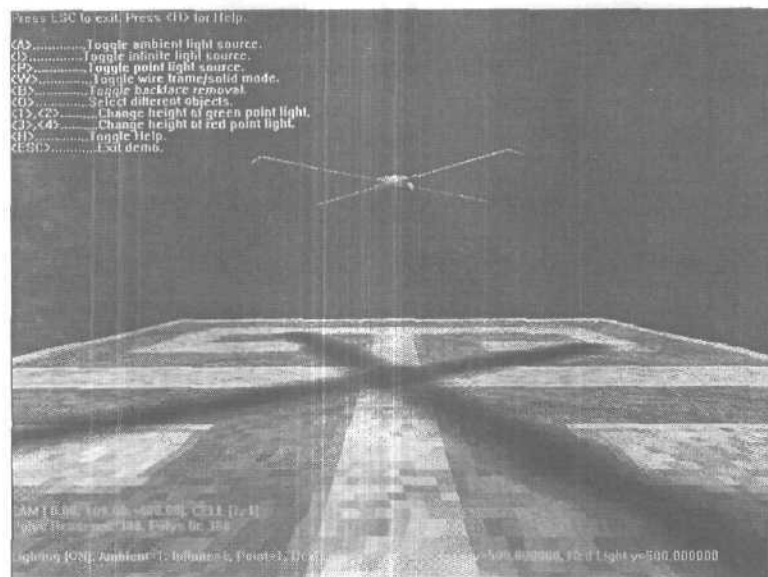


Рис. 14.34. Копия экрана демонстрационной программы с отображением освещения

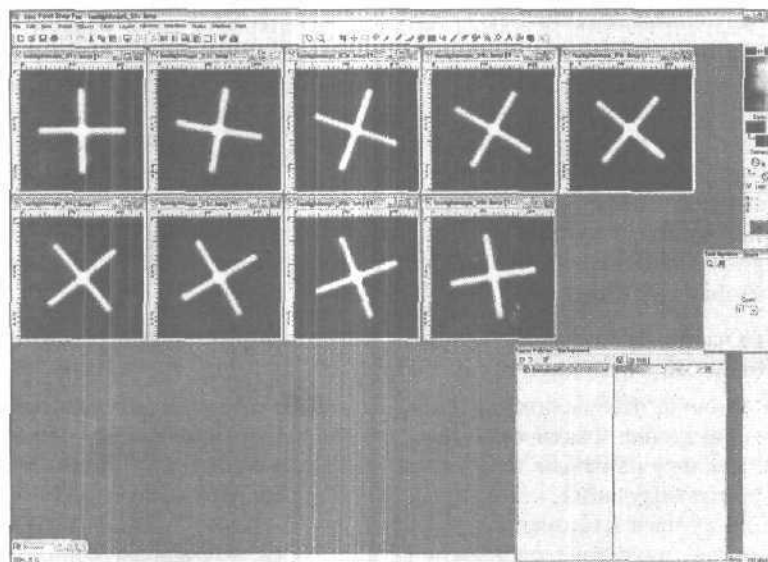


Рис. 14.35. Инвертированные карты

Затем я взял на себя смелость испытать себя в роли художника и немного их размыл, чтобы светлые части выглядели более расплывчатыми. Затем я поместил их в демонстрационный пример, и результаты вы можете найти на прилагаемом компакт-диске в файлах DEMO114\_6b.CPP\EXE. На рис. 14.36 показана копия экрана новой демонстрационной программы. Впечатляюще, не правда ли?



Рис. 14.36. Копия экрана демонстрационной программы с отображением темноты

## Получение специальных эффектов с помощью карт освещения

Прежде чем закончить эту главу, давайте поговорим о некоторых идеях и эффектах, которые можно достичь с **помощью** карт освещения. Прежде всего, самая большая проблема с картами освещения заключается в том, что они большей частью статические. Однако в действительности это не так важно, поскольку никто не запрещает вам анимировать карты освещения. Разумеется, важен и вопрос о **необходимой** памяти, но несколько областей с анимированными картами освещения сделать достаточно просто. Например, у вас может быть мерцающий свет (как в *Quake*), цветовые и другие эффекты, производимые с **помощью** множества карт освещения. Кроме того, предположим, что имеется большое помещение, разделенное на части. Значит, пока карты освещения для них не перекрываются, включение и выключение света в одной секции не влияет на освещенность других.

Другой эффект, достигаемый при помощи карт освещения, который действительно впечатляет, — это сложная анимация теней. Например, при открытии двери вы используете карту освещения, чтобы показать свет, падающий в темную комнату, и только при максимальном открытии двери вы включаете нормальное освещение. Можно также использовать карты освещения для проецирования анимации освещения на другие объекты. Например, можно взять пластмассовый прозрачный цилиндр и поместить в него источник света (конечно, все это делается в пространстве моделей). Вы вращаете модель в трехмерном пространстве, а затем используете текстуру в качестве карты **освещения** (на самом деле это называется *проекционной текстурой*, но идея остается той же, основанной на отображении освещения).

Наконец, используя математическую формулу для генерации текстур карт освещения, а затем текстуры — в качестве карт освещения, можно получить действительно впечатляющие облака, дым и другие эффекты, которые выглядят по-разному, но по сути представляют собой простую модуляцию освещения сцены.

Теперь, при наличии средств для создания теней в реальном масштабе времени и для отображения **освещения**, я не сомневаюсь в том, что вы полны энтузиазма осуществить свои самые смелые идеи. Однако имейте в виду, что каждый кадр и без того требует большого количества тактов процессора, так что хотя полные многоугольные проективные тени для каждого источника освещения и хорошо выглядят, лучше избежать макетов для **большого** количества объектов, особенно если они достаточно далеко удалены. В сомнительных случаях используйте отображение освещения, особенно для сцен внутри помещений.

## Оптимизация кода отображения освещения

Алгоритм кода отображения освещения достаточно оптимален, но понятно, что в коде всегда есть место оптимизации. Во-первых, вместо умножения можно использовать таблицу поиска. Но когерентность кэша может сильно замедлить выполнение, в то время как умножение требует только одного такта, так что я не вижу большого толка в использовании таблиц. Другая возможность заключается в использовании **SIMD-команд** и в выполнении одновременно четырех вычислений для отображения освещения. Это вполне выполнимо и именно то, что нужно сделать.

Но если на минутку отвлечься от кода модуляции, то реальной проблемой становится отображение света в игровом **процессоре**, которое не должно вызывать потери **скорости**, когда оно не используется (а при использовании не вызывает резкого снижения производительности). Мы уже рассмотрели технологию кэширования поверхностей для генерации текстур отображения **освещения**, и это лучшая оптимизация, которую я **могу** предложить. Стоит также использовать малые размеры текстур карт **освещения**, чтобы все они (или почти все) могли быть размещены в кэше, и тогда фаза модуляции текстуры может протекать очень быстро.

## Резюме

В этой главе было показано, как выход за рамки стандартного мышления может оказаться **самым** главным средством решения поставленных задач.

Мы узнали, что удивительные специальные эффекты — такие как тени, освещение, анимация и множество других эффектов — могут быть достигнуты с помощью методов кэширования поверхности. Запомните это и посмотрите, нельзя ли применить аналогичные методы и к другим аспектам ваших игр и игровых процессоров.

И наконец, эта глава выдвигает на первый план еще один аспект разработки **игр** — выбор соответствующих методов и инструментария. Как специалисты, которые программируют игры, мы должны уметь не только написать код, но и подумать о возможности привлечения художников и использования различных инструментальных средств. Если художник может сделать то же, что вы сделаете при помощи математики, так же хорошо с помощью искусства и затратить на это вдвое меньше времени — то кого будет интересовать, корректен ли полученный результат с точки зрения математики? Ключевым моментом является понимание, когда можно пойти на компромисс и “**подхимичить**”, а когда этого делать не стоит.



# ЧАСТЬ V

## Анимация, физическое моделирование и оптимизация

В этой части...

### Глава 15

Анимация, движение и обнаружение столкновений 1311

### Глава 16

Технологии оптимизации 1359



# ГЛАВА 15

## Анимация, движение и обнаружение столкновений

### В этой главе...

• Новый модуль игрового процессора	1312
• Введение в трехмерную анимацию	1312
• Формат QuakeII .MD2	1312
• Простая анимация без участия персонажа	1349
• Обнаружение трехмерных столкновений	1354

В этой главе мы в основном будем заниматься загрузкой и выводом анимированных трехмерных моделей из файла формата .MD2 игры *Quake II*. Мы также рассмотрим некоторые технологии перемещения в трехмерном пространстве и основные методы определения столкновений трехмерной модели с другой моделью и с окружающей средой. Вот краткое содержание главы:

- новый библиотечный модуль;
- введение в трехмерную анимацию;
- загрузка файлов формата .MD2;
- анимация файлов .MD2;
- вращательное и поступательное движение;
- параметрическое и криволинейное движение;
- использование сценариев движения;
- методы обнаружения столкновений и движения по местности.

## Новый модуль игрового процессора

В этой главе нам предстоит написать довольно много программного кода, поэтому целесообразно вынести его в отдельный библиотечный модуль, `T3DLIB13.CPP|H`. Чтобы откомпилировать любую программу из данной главы, нам понадобятся основной файл программы (`.CPP`), библиотечные файлы `DirectX (.LIB)` и два новых файла:

- `T3DLIB13.CPP` — исходный код C/C++ для теней и освещения
- `T3DLIB13.H` — соответствующий заголовочный файл

Само собой разумеется, необходимо подключить в проект файлы `T3DLIB1–12.CPP|H`.

## Введение в трехмерную анимацию

Основное внимание в данной главе мы уделим анимации, особенно анимации персонажей. Мне бы, конечно, хотелось рассказать о скелетной анимации и о файлах движения (по типу Biovision), но боюсь, что для этого придется ждать выхода следующего тома. Сейчас же мы рассмотрим анимацию и интерполяцию по ключевым кадрам. Анимация по ключевым кадрам выглядит довольно впечатляюще, и если ее достаточно для таких игр, как *Quake* и *Quake II*, то она явно пригодна и для наших экспериментальных целей. Таким образом, мы остановимся на трех основных моментах, связанных с анимацией: определение, загрузка и отображение моделей `.MD2`; расположение и перемещение трехмерных моделей и объектов; технология обнаружения столкновений для трехмерных моделей, представляющих персонажи игры.

Большая часть программного кода и примеров данной главы посвящены первой из поставленных нами задач: загрузка и вывод трехмерных моделей формата `.MD2` фирмы id Software. Это, пожалуй, самый сложный аспект данной главы. Оставшиеся же проблемы мы обсудим в основном теоретически, обращая свое внимание скорее на идеи, чем на их воплощение. Мне просто не удастся рассказать вам больше; к тому же мы рассмотрели уже столько задач и путей их решения, что вы всегда сумеете сориентироваться самостоятельно.

## Формат Quake II .MD2

Файл формата `.MD2`, который в дальнейшем мы будем обозначать просто `.MD2`, был разработан id Software для игры *Quake II* (рис. 15.1). В *Quake II* главным козырем оказалось аппаратное ускорение. *Quake II* символизирует конец эры использования программного ускорения в коммерческих продуктах. В *Quake II* используется множество современных технологий, и одна из них — анимация персонажей, улучшенная благодаря анимации по ключевым кадрам и интерполяции.

Хотя файлы `.MD2` и не поддерживают скелеты и прямую/обратную кинематику, тем не менее, данный формат сравнительно легко поддается анализу и довольно прост для понимания. Самым привлекательным является тот момент, что в Internet имеется множество файлов `.MD2`, а различных инструментов для работы с ними — еще больше. Таким образом, у вас не возникнет проблем с поиском моделей, инструментов и прочих приложений.

Я загрузил несколько бесплатных (для некоммерческих целей) моделей с Planet Quake. Однако автор каждой модели указывается, так что при желании сделать на них деньги вы должны связаться с разработчиком. А теперь поговорим о формате как таковом.

Начнем с того, что поиски хорошего описания файлов `.MD2` оказались намного труднее, чем чего бы то ни было другого в Internet. Я просмотрел каждую статью и книгу о

файлах .MD2, и в каждом случае авторы обязательно упускали какие-нибудь существенные детали, так что я постараюсь представить максимально исчерпывающую информацию о данном формате.

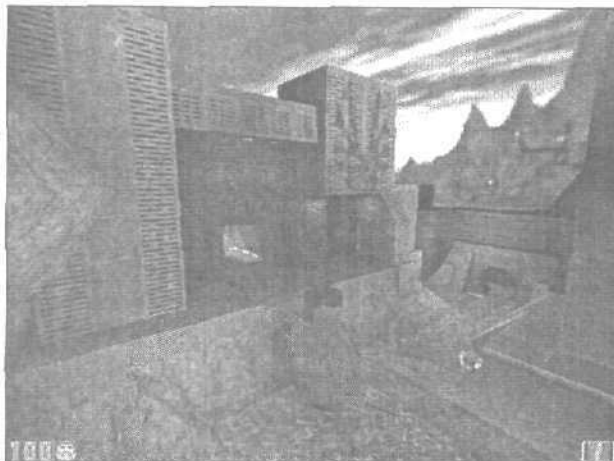


Рис. 15.1. Копия экрана игры *Quake II*

**НА ЗАМЕТКУ**

Один из лучших ресурсов, где можно найти модели *Quake*, — Planet Quake:

<http://www.planetquake.com/polycount/>

Инструментарий для работы с файлами .MD2 можно найти по адресу

<http://www.planetquake.com/polycount/resources/quake2/tools.shtml>

а техническую спецификацию формата — по адресу

<http://www.planetquake.com/polycount/resources/quake2/q2frameslist.shtml>

Давайте начнем с общего описания, а затем перейдем к деталям. Файл формата .MD2 состоит из двух основных частей: заголовок и данные, как показано на рис. 15.2.

- Заголовок: содержит описание модели, количество многоугольников, вершин, деталей анимации и т.д.
- Данные: собственно информация о модели в виде набора многоугольников, включая все данные об этих многоугольниках, вершинах и текстурах.

**СОВЕТ**

Все многоугольники в файле .MD2 — треугольники, и только треугольники!

Заголовок файла .MD2 — это стандартный заголовок, ничего необычного в нем нет, а вот данные довольно интересны. Так, например, данные, касающиеся вершин (которыми мы займемся в свое время), определяют на самом деле несколько анимационных кадров, а не один. В .MD2-файлах возможно задание до 198 анимационных кадров, которые имеют номера от 0 до 197.

**ВНИМАНИЕ**

Вам могут встретиться модели, содержащие 199, 200 и даже более анимационных кадров, что является отклонением от исходного формата. Тем не менее, оригинальная спецификация от id Software определяет лишь 198 кадров. Однако это касается только анимационных кадров, поскольку для задания различных действий используются индексы.

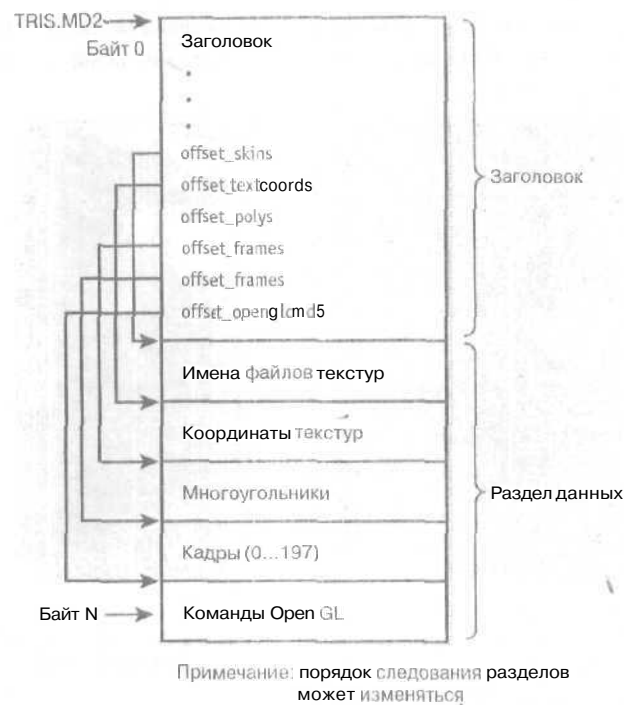


Рис. 15.2. Структура файла Quake II .MD2

В анимационных кадрах задается множество многоугольников, состоящее из набора вершин в пространстве модели с началом координат в точке  $(0,0,0)$  в правой системе координат, упорядоченных против часовой стрелки (это будет важно для нас позже, а сейчас просто имейте это в виду). Вершины определяют вид объекта для каждого кадра анимации, и каждый из этих кадров может входить в отдельную анимационную последовательность (объект ходит, бежит, прыгает и т.д.). Мы, конечно же, рассмотрим все это в деталях, но сейчас просто отметим, что в начале файла есть заголовок, который задает модель, и, основываясь на информации из заголовка, вы загружаете данные кадра, которые представляют собой обычный поток бинарных данных.

Анимация осуществляется последовательным прохождением по кадрам анимации. Например, анимация "стоящего" объекта задается последовательностью кадров 0—39 таким образом, что последовательно отображая эти кадры, мы получим анимацию объекта в состоянии покоя. Однако следует заметить, что эти кадры являются ключевыми, т.е. при желании вы можете использовать линейную интерполяцию между кадрами для получения более плавной анимации. Например, вы решили провести интерполяцию с коэффициентом 0.25 от кадра  $i$  до кадра  $i+1$ . При этом будет осуществлен плавный переход координат вершин от кадра  $i$  до кадра  $i+1$  с интерполяционными значениями 0, 0.25, 0.50, 0.75, 1.0.

Следующий важный момент — это текстурная карта, или *скин* (букв. кожа), как называют ее парни из idSoftware. Рис. 15.3 демонстрирует стандартный вид текстур для .MD2-модели (Warhammer из Polycount, разработанный Джо Вудреллом (Joe Woodrell)). Текстура представляет собой 256-цветный рисунок размером 256x256 пикселей, в большинстве случаев в формате .eps. Таким образом, нам придется преобразовывать текстуры в 16-битовый цвет и формат .BMP.

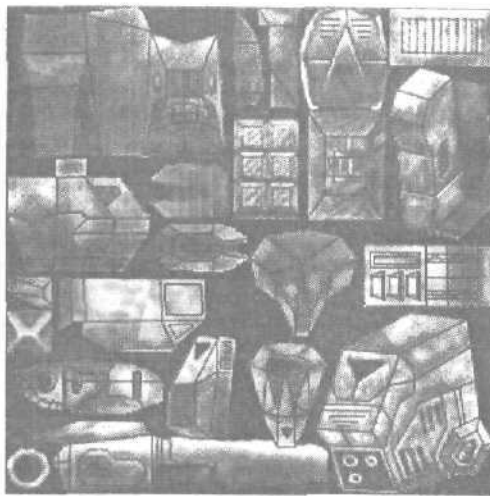


Рис. 15.3. Текстура из Quake II (скин)

В любом случае, назначение скина — быть использованным в качестве текстуры для модели. Каждая модель может иметь много скинов (для различных стилей и цветов). Интересно отметить, что скин не является частью .MD2 файла. В файле присутствуют только лишь ссылки на них, а сами скины размещаются в отдельных файлах в каталоге Quake II. Таким образом, в процессе анализа заголовка .MD2 файла можно обнаружить наличие, например, трех скинов для модели (скажем, синий, камуфляжный и кроваво-красный). В таком случае нужно будет воспользоваться указанным путем к файлу .eps для загрузки текстуры. Позже мы к этому вернемся.

## Заголовок .MD2

Теперь давайте рассмотрим заголовочный раздел .MD2-файла более детально. В-первых, большинство моделей, которые вы найдете в Internet, окажутся после распаковки в одном каталоге. Обычно имеется несколько файлов, относящихся к модели, например, описание оружия или звука. Нам же интересны лишь модели как таковые и их скины, поэтому сконцентрируем свое внимание именно на них. Когда вы загрузите модель из Internet или просмотрите одну из находящихся на прилагаемом компакт-диске, вы сможете обнаружить, по крайней мере, следующие два файла:

- **TRIS.MD2** — это файл, в котором содержатся данные, описывающие модель персонажа;
- **WEAPON.MD2** — этого файла может и не быть, но обычно он присутствует. В нем содержится описание моделей оружия в формате .MD2. Вы можете загрузить его таким же образом, как и модели персонажей.

Далее вы обнаружите .eps-файлы с размещенными в них 256-цветными текстурами размером 256x256 пикселей. Просто просмотрите их с помощью любой подходящей программы (Paint Shop Pro, Photoshop и т.п.). Изображения, которые вы увидите, в большинстве своем будут скинами для текстурирования модели. Они имеют размер 256x256, поскольку обе координаты текстуры ( $u, v$ ) в разделе данных файла .MD2 принимают значения от 0 до 255. Однако чтобы воспользоваться любой из этих текстур, нам необходимо преобразовать их в 16-битовый формат и сохранить как .BMP-файл (несмотря на то, что у нас есть загрузчик файлов .eps, формат .BMP все же значительно проще).

Вы также обнаружите множество маленьких пиктограмм и .WAV-файлов, содержащих звуковые эффекты. Впрочем, пусть они вас не беспокоят. В действительности, все, что нам понадобится, — это файл **TRIS.MD2** и одна или несколько скинов-текстур, преобразованных в формат **.BMP**. Обычно я действую следующим образом. Я загружаю модель **.MD2** с Web-узла Polyscount или какого-нибудь другого и распаковываю ее в каталог. Затем я просматриваю изображения в каталоге в поисках скинов. Найдя пару понравившихся, я увеличиваю глубину их цвета до 16 битов, поднимаю яркость на 10-20% и сохраняю в формате **.BMP**. Название файла при этом не имеет **значения**, поскольку мы собираемся заменить названия **скинов-текстур**, записанных в заголовке, для использования наших собственных текстур. При этом все будет работать корректно, поскольку координаты текстуры (u, v) находятся в пространстве **256×256**, а текстуры остаются по сути теми же.

#### СОВЕТ

Изначально поддержка текстур спроектирована таким образом, что они загружаются из каталога **.EXE-файла**, т.е. нам не надо извлекать имя файла — можно просто использовать имя **.BMP-версии текстуры**.

Конечно, мы не можем применять произвольные текстуры: надо использовать одну из реально **существующих** текстур модели во избежание нестыковки координат текстуры. Мы можем всего лишь упростить доступ к текстурам на диске, **перемещая и переименовывая** их файлы.

Итак, поскольку теперь вы знаете, что нужно искать на жестком диске при загрузке **.MD2** файла, давайте посмотрим на заголовок и данные. Во-первых, запомните, что файл **.MD2** полностью бинарный, и данные должны считываться как поток байтов. Ниже представлено определение заголовка, который всегда идет в **.MD2** первым, начиная с байта 0.

// Заголовок файла **.MD2** для **Quake II**

```
typedef struct MD2_HEADER_TYP
{
    int identifier;    // Идентифицирует тип файла,
                     // должно быть "IDP2"
    int version;       // Номер версии, должен быть равен 8
    int skin_width;    // Ширина текстурной карты,
                     // использованной для создания скина
    int skin_height;   // Высота текстурной карты,
                     // использованной для скина
    int framesize;     // Число байтов в одном кадре анимации
    int num_skins;     // Общее число скинов, указанных в
                     // имени файла ASCII и доступных
                     // для загрузки, если файлы найдены
    int num_verts;     // Число вершин в каждом кадре модели;
                     // число вершин в каждом кадре всегда
                     // одно и то же
    int num_textcoords; // Общее число координат текстуры
                     // в целом файле; может превышать число
                     // вершин
    int num_polys;     // Количество многоугольников на одну
                     // модель, или на один кадр анимации
    int num_openGLcmds; // Число команд OpenGL, которые
                     // могут помочь при оптимизации
                     // отображения (мы не будем их
                     // использовать)
    int num_frames;    // общее число анимационных кадров
}
```

```

// смещения в байтах для каждого компонента

int offset_skins;      // Смещение в байтах от начала файла
                       // до массива скинов, который содержит
                       // имя файла для каждого скина; запись
                       // каждого имени файла имеет размер 64
                       // байта
int offset_textcoords; // Смещение в байтах от начала
                       // файла до массива координат текстур
int offset_polys;      // Смещение в байтах от начала файла
                       // до сетки многоугольников
int offset_frames;     // Смещение в байтах от начала файла
                       // до данных вершин для каждого кадра
int offset_openGLcmds; // Смещение в байтах от начала
                       // файла к командам OpenGL
int offset_end;        // Смещение конца файла в байтах от
                       // начала файла
} MD2_HEADER, *MD2_HEADER_PTR;

```

В зависимости от того, где вы встретитесь с описанием файла .MD2 и его заголовка, вы встретите немного отличные имена областей, но большей частью каждый пытается придерживаться имен из оригинального определения id Software. Однако типы данных и их порядок должны быть идентичными предшествующей структуре, в противном случае вы не сможете обращаться к данным заголовка при загрузке файла TRIS.MD2. Давайте рассмотрим каждое поле отдельно.

int identifier — это поле определяет тип файла .MD2, и его значение должно быть равно 'TDP2' с обратным порядком байтов. Другими словами, на персональном компьютере с процессором Intel вы должны получить следующее значение.

```
#define MD2_MAGIC_NUM (('T' << 8) + ('P' << 16) + ('2' << 24))
```

int version — номер версии, и он всегда должен быть равен 8. Ниже представлен макрос для данной проверки.

```
#define MD2_VERSION 8
```

Если идентификатор или версия указаны неверно, то вряд ли вы сможете загрузить что-то стоящее из такого файла.

int skin\_width — ширина в пикселях текстурной карты, используемой для создания скина модели. Обычно это 256, но ширина может быть и другой, так что если вы действительно хотите разработать надежный код, примите это во внимание.

int skin\_height — высота в пикселях текстурной карты, используемой для создания скина модели. Обычно это 256, но высота может быть и другой, так что если вы действительно хотите разработать надежный код, примите это во внимание.

Заметим, что все текстурные карты для конкретной модели будут иметь скины одинакового размера, так что вам никогда не придется беспокоиться об изменении размера текстур по ходу дела. В 99% случаев текстуры имеют размер 256x256.

int framesize — довольно хитрое поле. Как показано на рис. 15.4, это размер в байтах единичного кадра анимации модели. Каждый кадр состоит из заголовка и информации о каждой из вершин модели. Мы вскоре вернемся к этому вопросу, а пока просто запомните, что единичный кадр состоит из всех вершин модели и заголовка, который говорит нам о том, как нужно масштабировать и позиционировать эти вершины. В большинстве случаев для файлов .MD2 от id Software таких кадров 198.

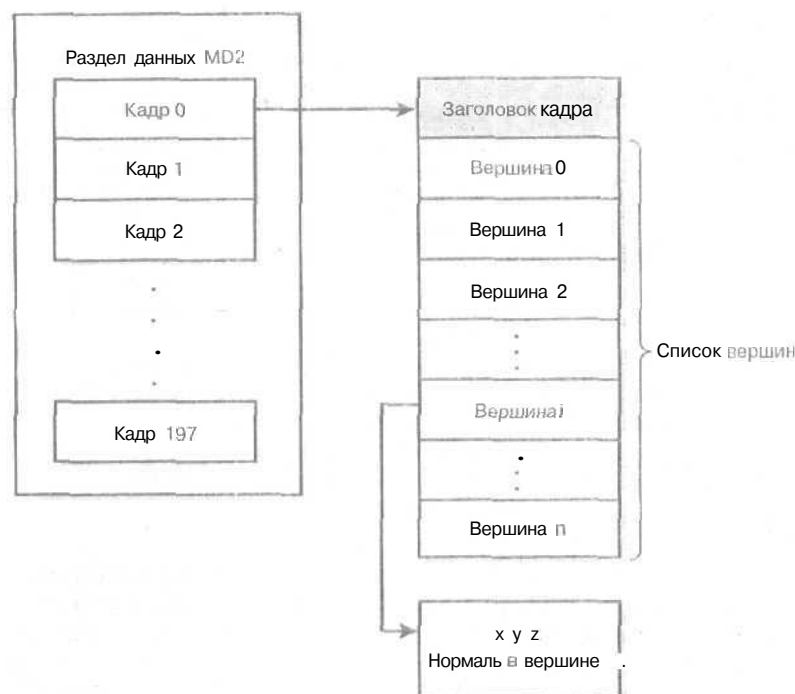


Рис. 15.4. Структура отдельного анимационного кадра

`int num_skins` — это поле задает общее число текстур, определяемых именами файлов, находящихся на диске вместе с файлами моделей. Имя каждой текстуры находится в массиве строк в конце заголовка (в некоторой определенной позиции). Длина имени каждой текстуры — 64 байта. Незанятое именем пространство поля заполняется нулями. Модель может иметь одно или несколько имен текстур. В большинстве случаев эти имена используют информацию о путях, специфичную для *Quake II*, поэтому мы игнорируем их и, просмотрев вручную, переименовываем, а затем, при загрузке модели, передаем соответствующую строку с путем и именем файла (в 16-битовом .BMP-формате).

`int num_verts` — здесь задается количество вершин в единичном кадре модели (не общее число вершин в файле, а именно их количество в единичном кадре анимации). Это очень важно. Поскольку модель состоит всегда из одних и тех же многоугольников, число вершин всегда одно и то же от кадра к кадру. Соответственно, единственное, что изменяется от кадра к кадру, — это положение вершин. Таким образом создается анимация в формате .MD2.

`int num_textcoords` — это поле определяет общее число координат текстур в файле модели. Здесь будьте осторожны: вы можете решить, что координаты текстуры нужны для каждой вершины. Но это вовсе не обязательно так. Вспомните, что координаты текстуры — это пара координат ( $u, v$ ) в пространстве текстуры. Однако часто текстурных координат может быть больше или меньше, чем вершин, поскольку в одни и те же координаты текстуры могут отображаться несколько вершин. В большинстве случаев имеется несколько сот текстурных координат. Кроме того, есть только одно множество этих координат, совместно используемое всеми кадрами анимации (рис. 15.5).

`int num_polys` — в этом поле указывается количество многоугольников модели (или в одном кадре анимации). Это число всегда одно и то же для каждого кадра, так что если модель имеет 800 многоугольников, это означает, что и каждый кадр анимации имеет ровно 800 многоугольников.

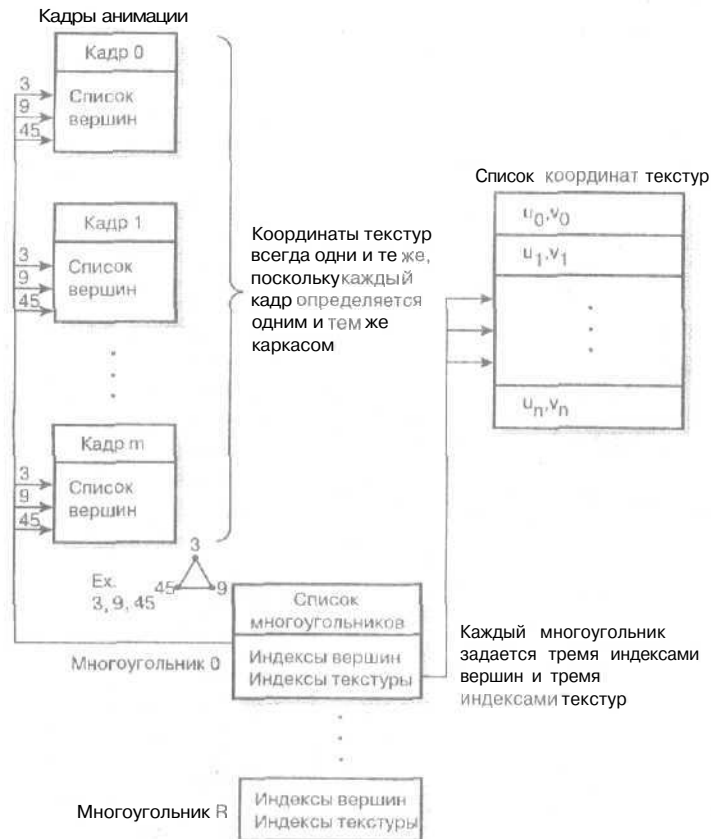


Рис. 15.5. Координаты текстур используются всеми кадрами анимации

`int num_openGLcmds` — это поле мы игнорируем. Скажу лишь, что здесь задается количество команд OpenGL, которые могут помочь в оптимизации аппаратной визуализации.

`int num_frames` — определяет общее количество кадров анимации. Для большинства .MD2 моделей оно равно 198.

Далее следуют поля, в которых хранятся смещения реальных данных относительно начала файла. Имейте в виду, что .MD2-файлы — бинарные. Смещения указывают, сколько байтов от начала файла надо пропустить для доступа к определенным группам данных. Затем вы можете получить доступ к данным при помощи соответствующих структур (рис. 15.6.)

`int offset_skins` — смещение массива скинов, в котором хранятся имена файлов каждой текстуры. Каждая запись имеет размер 64 байта.

`int offset_textcoords` — смещение массива координат текстур. Каждая координата определяется следующим образом.

```
// Координаты текстуры (u, v)
typedef struct MD2_TEXTCOORD_TYP
{
    short u,v; // Координаты текстуры
} MD2_TEXTCOORD, *MD2_TEXTCOORD_PTR;
```

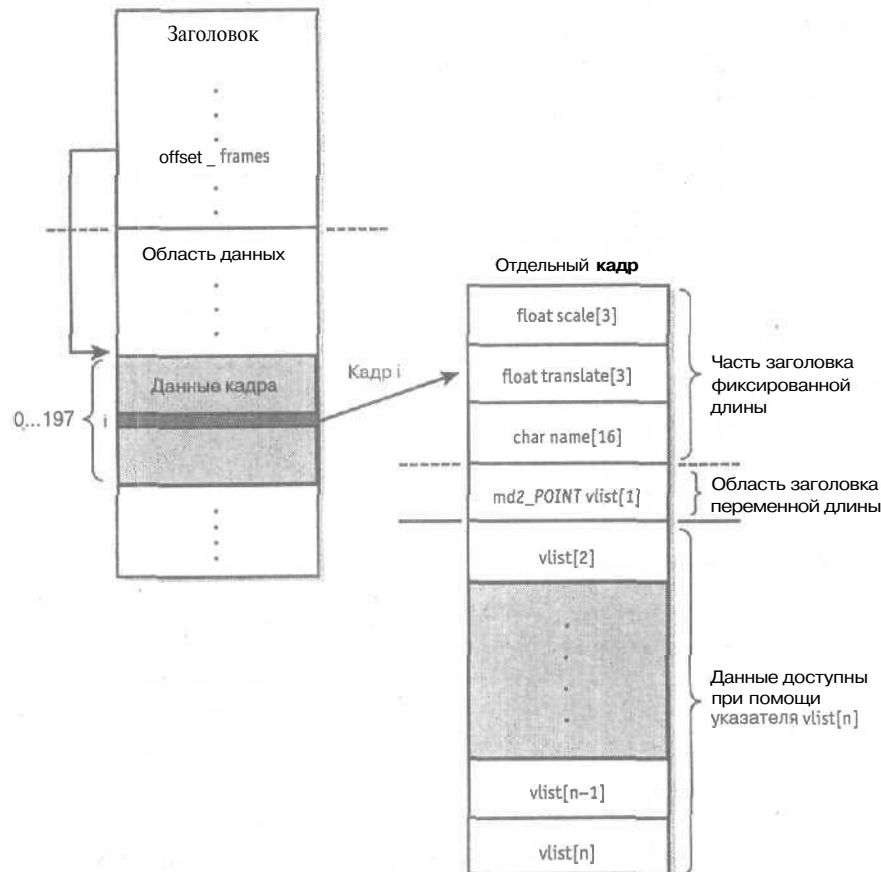


Рис. 15.6. Шаблон структуры данных переменного размера

Таким образом, каждая координата имеет размер 32 бита, или 4 байта. Мы используем поле `num_textcoords` вместе с предыдущей структурой для доступа к отдельным координатам.

`int offset_polys` — смещение данных о каркасе. Каждый составляющий многоугольник определен как множество координат образующих его вершин. А теперь будьте внимательны. Предположим, что многоугольник 0 имеет координаты вершин (12,56,99). Это означает, что многоугольник 0 всегда имеет вершины (12,56,99) — не имеет значения, какое множество вершин или кадр используется. Итак, все, что нам надо, — это единственное определение многоугольника для всей модели, которое связывает многоугольник с индексами вершин. Для реализации анимации выполняется обмен множеств вершин, но при этом используются одни и те же индексы вершин (рис. 15.7).

Формат данных для определения каждого многоугольника выглядит следующим образом.

```
// Структура данных для единичного многоугольника md2
// (треугольника); содержит 3 вершины и 3 координаты
// текстуры, которые представляют собой индексы
typedef struct MD2_POLY_TYP
{
    unsigned short vindex[3]; // Индексы вершин
    unsigned short tindex[3]; // Индексы текстуры
} MD2_POLY, *MD2_POLY_PTR;
```

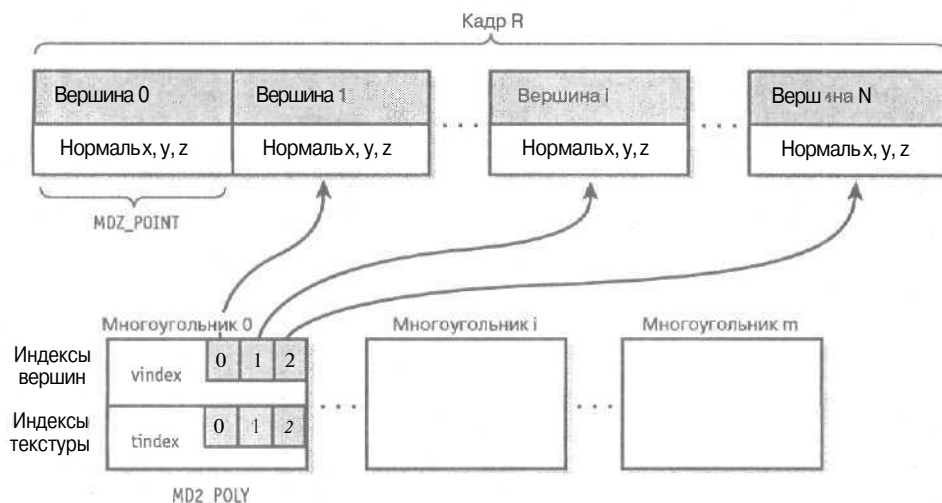


Рис. 15.7. Связь между многоугольниками и вершинами

Список многоугольников представляет собой массив многоугольников, каждый из которых содержит 6 элементов типа unsigned short. Первые три определяют индексы вершин, а вторые три — индексы координат текстур. Таким образом, многоугольник определяется косвенно, как показано на рис. 15.8.

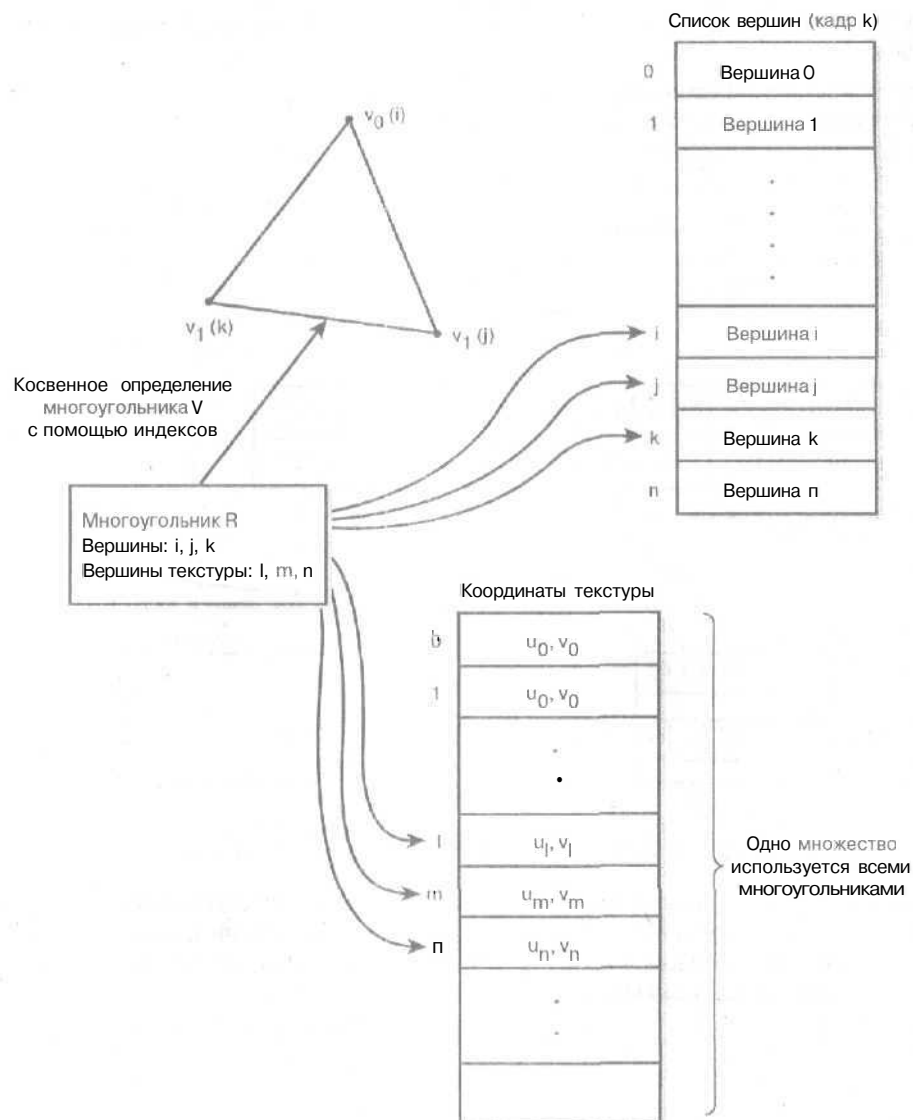


Рис. 15.8. Косвенное определение многоугольников

`int offset_frames`— смещение данных о вершинах для каждого кадра. Это, пожалуй, наиболее запутанные данные, поскольку не совсем понятно, из чего состоит каждый кадр. Поэтому давайте попробуем разобраться с этим вопросом более подробно. Каждый кадр состоит из небольшого заголовка и массива вершин, которые определяют кадр анимации (рис. 15.9).

Заголовок кадра определяется следующим образом.

```
// Кадр формата md2 с блоком заголовка, который описывает,
// как надо масштабировать и перемещать вершины модели.
// Далее следует массив данных, к которым можно обращаться
```

```
// как к элементам массива благодаря определению vlist
typedef struct MD2_FRAME_TYP
{
    float scale[3];      // Коэффициенты масштабирования
                        // вершин в кадре по осям x, y, z
    float translate[3];  // Коэффициенты перемещения
                        // вершин в кадре по осям x, y, z
    char name[16];       // ASCII имя модели
    MD2_POINT vlist[1];  // Начало массива вершин
}

```

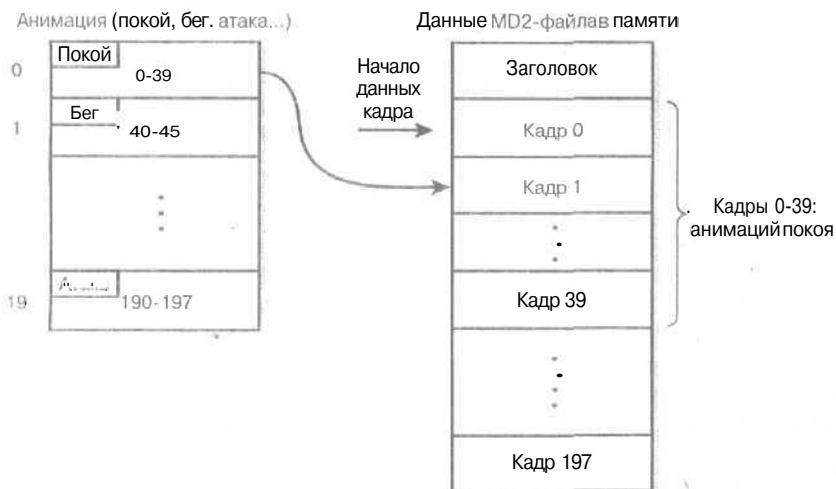


Рис. 15.9. Массив кадров анимации

Фактические данные о вершинах следуют за заголовком, но поговорим пока о самом заголовке. Он имеет четыре поля.

`float scale[3]` — это три коэффициента масштабирования по осям  $x$ ,  $y$  и  $z$ , так что когда вы загружаете кадр .MD2, вы должны масштабировать с помощью этих коэффициентов каждую вершину. Коэффициент масштабирования по оси  $x$  хранится в `scale[0]`, по оси  $y$  — в `scale[1]` и по оси  $z$  — в `scale[2]`. Есть две причины для использования коэффициентов масштабирования: во-первых, с целью экономии пространства каждая вершина сжимается до 8 бит (напоминает вейвлет-сжатие), так что вы должны преобразовать каждую вершину к ее нормальному размеру; во-вторых, для собственно анимации. Представьте, что вы хотите масштабировать кадр как часть анимации — вот вам и подходящий способ реализации этой идеи.

`float translate[3]` — три коэффициента переноса по осям  $x$ ,  $y$  и  $z$ , так что когда вы загружаете файл .MD2, вы должны с их помощью преобразовать каждую вершину. Коэффициент переноса по оси  $x$  хранится в `translate[0]`, по оси  $y$  — в `translate[1]` и по оси  $z$  — в `translate[2]`. Причина использования коэффициентов переноса такая же, как и в предыдущем случае — экономия пространства на диске, т.к. каждая вершина занимает всего 8 бит, но, что более важно, это делается для перемещения каждого кадра в пространстве модели, что, собственно, и создает анимацию. Например, таким образом достигается эффект "прыжка": переносом координат модели, что гораздо лучше, чем перемещение положения в мировых координатах и преобразования локальных координат в мировые.

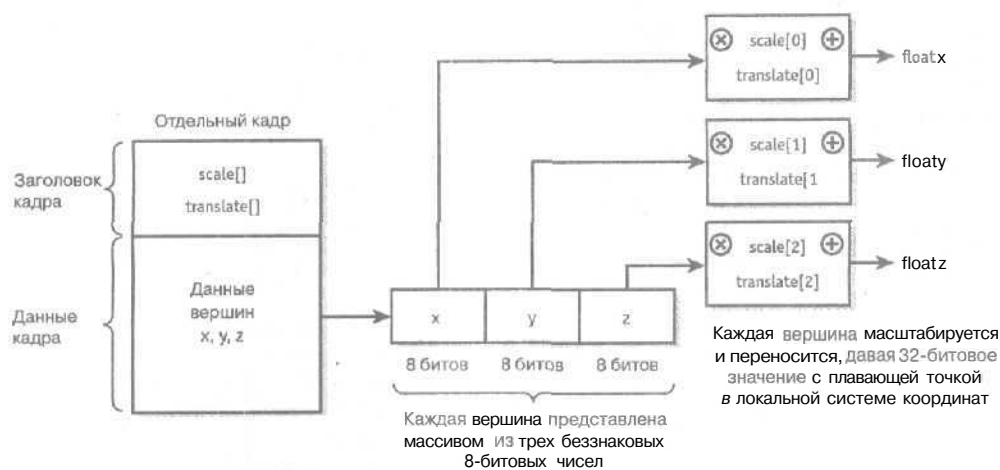


Рис. 15.10. Преобразования вершин

На рис. 15.10 показаны преобразования, которые определены в заголовке кадра и выполняются с каждой из вершин; а вот как выглядит соответствующий псевдокод для заданной вершины *v* и массивов *scale[]* и *translate[]*.

```
v.x = v.x * scale[0] + translate[0];
v.y = v.y * scale[1] + translate[1];
v.z = v.z * scale[2] + translate[2];
```

`char name[16]` — это обычное текстовое имя модели, например, “TerminatorT1000”.

После этого, наконец, начинаются настоящие данные,

`MD2_POINT vlist[1]` — это указатель на массив вершин. Однако мы не знаем, сколько их всего, поэтому здесь используется трюк с определением массива размером 1. Если вы обратитесь к элементу `vlist[2]`, компилятор позволит вам сделать это (правда, вы должны убедиться, что по этому адресу есть доступные данные, иначе получите ошибку доступа к памяти). Это старый испытанный прием. Таким образом, список вершин `vlist[]` — это список вершин кадра анимации. Таких вершин всегда ровно `num_verts`, а их формат выглядит следующим образом.

```
// Единичная точна в модели md2, содержит 8-битовые
// масштабированные координаты x,y,z и индекс нормали
typedef struct MD2_POINT_TYP
{
    unsigned char v[3]; // Вершина x,y,z в сжатом формате
    unsigned char normal_index; // Индекс в таблице нормалей
} MD2_POINT, *MD2_POINT_PTR;
```

Обратите внимание на то, что координаты *x*, *y*, *z* каждой точки определены как `unsigned char`. Теперь вы видите, почему так необходимо масштабирование каждого кадра. Каждая вершина состоит из 3 байтов, за которыми следует байт `normal_index`. Эта переменная используется для еще одной схемы сжатия. В каждом кадре у каждой вершины имеется **нормаль**, как показано на рис. 15.11.

Нормаль располагается в пространстве модели, но указывается как индекс в таблице предвычисленных нормалей. Мы не будем использовать эти значения, вычисляя нормали только на основании информации о многоугольнике, но это еще один пример того, как можно достичь экономии памяти за счет дополнительного уровня косвенности.

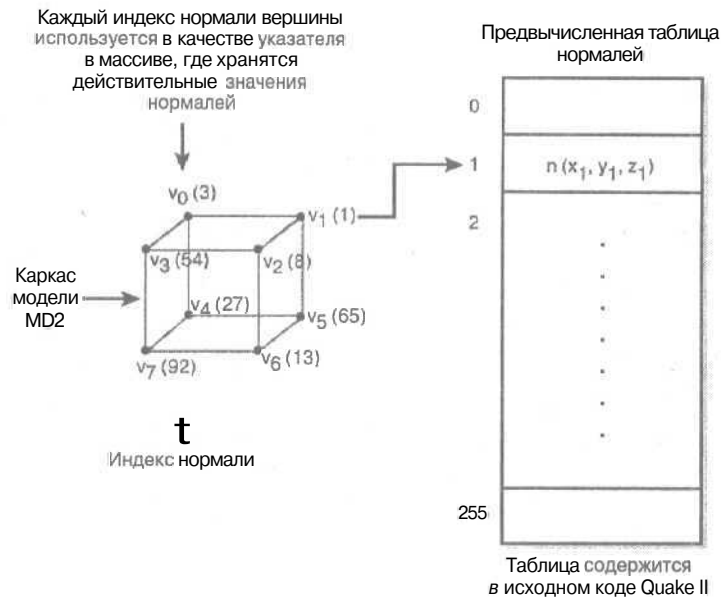


Рис. 15. И. Нормали освещения в Quake II

Итак, в начале данных кадров со смещением `offset_frames` находится массив кадров (всего их 198). Каждый кадр состоит из заголовка и набора вершин. В заголовке содержатся коэффициенты масштабирования и переноса вместе с именем кадра, после которых следуют данные о вершинах кадра. Данные состоят из массива 4-байтовых величин, первые три байта которых — координаты  $x$ ,  $y$ ,  $z$  вершины в пространстве модели, а последняя величина — это индекс в таблице нормалей вершин из 256 элементов (которая нами не поддерживается).

`int offset_openglcmds` — смещение команд OpenGL. Поскольку мы не используем OpenGL, то и останавливаться на этом не будем.

`int offset_end` — смещение конца файла. Конечно, мы можем просто запросить размер файла у операционной системы, но в принципе эта величина хранится в самом файле.

А теперь давайте займемся программированием и напомним код для загрузки файлов `.MD2`.

## Загрузка `.MD2`-файлов Quake II

Как вы, наверное, помните, вначале мы думали, что структура `OBJECT4DV2` будет использоваться и для поддержки анимации, и написали немало кода для поддержки множественных кадров. Однако `OBJECT4DV2` — это всего лишь класс-контейнер, который визуализирует сетку — так почему он должен поддерживать анимацию, `.MD2` файлы или что-то еще? Лучшим выходом будет просто написать дополнительные функции заполнения `OBJECT4DV2` данными о многоугольниках, а затем передать этот объект процессору визуализации, используя при этом имеющееся программное обеспечение. На этом я и решил остановиться, так что вся поддержка анимации, написанная нами, стала упражнением для освоения клавиатуры :-).

Общий план решения проблемы загрузки и визуализации `.MD2`-файлов таков.

1. Загрузить файл `.MD2` и преобразовать его в более доступный формат, из которого могут быть извлечены кадры и загружены в `OBJECT4DV2`.

2. Написать пару вспомогательных функций для подготовки объектов OBJECT4DV2 для использования в качестве “носителей” кадров .MD2.
3. Написать функцию извлечения данного кадра из анимации модели .MD2, загрузить его и OBJECT4DV2, чтобы после этого его можно было передать уже разработанному игровому процессору для визуализации.

Сначала я думал, что смогу извлекать кадры непосредственно из .MD2-файла в память, но это предполагает масштабирование кадра, перенос и другие действия. Кроме того, эту идею делает неприемлемой наличие упакованных структур и косвенности в структурах данных. Лучше создать еще один контейнер для хранения данных .MD2, но такой, который использовал бы имеющиеся у нас структуры данных для точек, вершин, текстур и прочего. Имея доступ к такому классу-контейнеру, мы могли бы очень быстро извлекать из него кадры.

Здесь самое главное — это скорость. Мы хотим визуализировать кадры без задержек и без накладных расходов на переключение между кадрами. Мы должны также немного подумать о наших дальнейших действиях при разработке контейнера для поддержки анимации, движения и общей информации о состоянии объекта. В конечном счете я пришел к такой структуре.

```
typedef struct MD2_CONTAINER_TYP
```

```
1
{
    int state;           // Состояние модели
    int attr;            // Атрибуты модели
    int color;           // Основной цвет при отсутствии
                        // текстуры
    int num_frames;      // Количество кадров в модели
    int num_polys;       // Количество многоугольников
    int num_verts;       // Количество вершин
    int num_textcoords;  // Количество координат текстуры

    BITMAP_IMAGE_PTR skin; // Указатель на текстуру модели

    MD2_POLY_PTR polys;   // Указатель на список
                        // многоугольников
    VECTOR3D_PTR vlist;   // Указатель на список координат
                        // вершин
    VECTOR2D_PTR tlist;   // Указатель на список координат
                        // текстуры

    VECTOR4D world_pos;   // Положение объекта
    VECTOR4D vel;         // Скорость объекта

    int ivars[8];         // Целочисленные переменные
    float fvars[8];       // Вещественные переменные
    int counters[8];      // Счетчики общего назначения
    void *link;           // Указатель общего назначения

    int anim_state;       // Состояние анимации
    int anim_counter;     // Счетчик анимации общего
                        // назначения
    int anim_mode;        // Режим - одиночный кадр, цикл
    int anim_speed;       // Чем меньше это число, тем выше
```

```

        // скорость анимации
int anim_complete; // Флаг, свидетельствующий о
                    // завершении анимации
float curr_frame;   // Текущий кадр анимации

} MD2_CONTAINER, *MD2_CONTAINER_PTR;

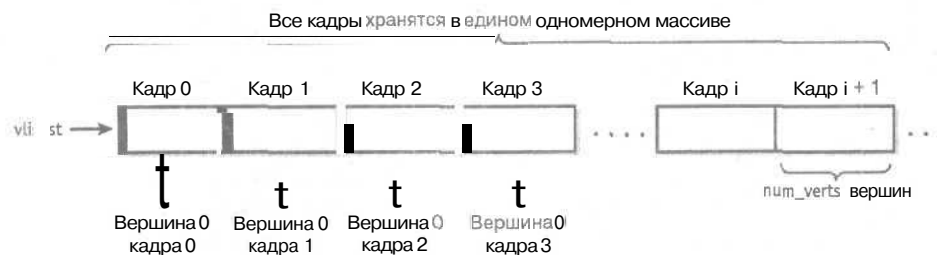
```

Все поля ясно описаны, и многие из них представляют собой копии данных из заголовка .MD2. Тем не менее, давайте рассмотрим их подробнее.

- `int state` — состояние модели: жив ли персонаж, мертв и так далее.
- `int attr` — атрибуты многоугольников модели, указывающие, каким образом они должны быть визуализированы (например атрибуты затенения). Нам требуются эти данные, поскольку формат .MD2 не несет никакой информации об освещении и затенении, и мы сами должны определить, каким образом визуализировать модель. Это поле используется для хранения данных флагов и позже вносится в OBJECT4DV2 для каждого многоугольника. В большинстве случаев, атрибуты будут следующими.  
(POLY4DV2\_ATTR\_RGB16 | POLY4DV2\_ATTR\_SHADE\_MODE\_FLAT |  
POLY4DV2\_ATTR\_SHADE\_MODE\_TEXTURE)
- `int color` — основной цвет в случае отсутствия текстуры. Однако модель, выполненная в одном-единственном цвете, скучна, поэтому в большинстве случаев эта величина будет устанавливаться равной RGB(255,255,255) для обеспечения максимальной отражательной способности текстуры.
- `int num_frames` — общее количество кадров модели .MD2 для всей анимации (обычно 198). Данный элемент копируется из заголовка файла .MD2.
- `int num_polys` — общее количество многоугольников, которые составляют единственный анимационный кадр. Оно такое же, как в заголовке файла .MD2, и всегда одно и то же для каждого кадра анимации, потому что сетка многоугольников — это шаблон, и единственное, что может изменяться от кадра к кадру, — ее вершины.
- `int num_verts` — число вершин в каждом кадре анимации (обычно в анимационном кадре .MD2 их сотни). Оно всегда одно и то же для всех кадров анимации. Единственное, что изменяется с целью оживить модель, — это положение вершин.
- `int num_textcoords` — количество координат текстуры для всех вершин. Каждый кадр анимации обращается к координатам текстуры по индексу в массиве координат текстур. В большинстве случаев в модели .MD2 есть несколько сотен координат текстур.
- `BITMAP_IMAGE_PTR skin` — указатель на текстуру модели. Этот битовый массив получается путем преобразования 256-цветной текстуры размером 256x256 из каталога модели в 16-битовый .BMP-формат.

Следующие поля определяют, где именно хранятся геометрические данные, но в отличие от .MD2-файла, эти данные готовы к визуализации — с ними выполнены необходимые преобразования масштабирования, переноса и г.д. Кроме того, все они сохранены с использованием разработанных нами структур данных, так что игровому процессору не требуются никакие дополнительные действия для работы с ними.

- **MD2\_POLY\_PTR polys** — указатель на список многоугольников, который представляет собой массив структур. Каждая структура является записью, содержащей индексы вершин и индексы текстуры для каждой вершины (так же, как и в **.MD2-файле**). В данном случае нет необходимости в каком-то ином формате, потому что это по существу косвенный массив.



Вершина  $j$  кадра  $i$ :  
 $vlist[i * num\_verts + j]$

Рис. 15.12. Список вершин в последовательности кадров

- **VECTOR3D\_PTR vList** — указатель на список координат вершин. Каждая вершина представляет собой стандартный **VECTOR3D**. В этом массиве для каждого кадра указаны все вершины, так что в нем хранятся тысячи вершин. Данные хранятся последовательно, как показано на рис. 15.12. Каждый кадр состоит из **num\_verts** вершин, каждая из которых имеет тип **VECTOR3D**. Никакой другой структуры не налагается, так что для получения доступа к конкретному кадру анимации и вершинам этого кадра надо умножить номер кадра на **num\_verts** и использовать это значение как базовый индекс.
- **VECTOR2D\_PTR tlist** — указатель на список координат текстур. Каждая координата текстуры хранится как число с плавающей точкой в составе **VECTOR2D**. В этом массиве хранится полный список координат текстур, состоящий из **num\_textcoords** элементов.

Следующие два поля используются для позиционирования и перемещения модели в трехмерном пространстве:

- **VECTOR4D world\_pos** — положение модели в мировых координатах;
- **VECTOR4D vel** — скорость модели.

Следующие вспомогательные поля нужны для хранения информации о состоянии, работы логики игры и т.п.

- **int ivars[8]** — несколько целых переменных общего назначения;
- **float fvars[8]** — несколько вещественных переменных общего назначения;
- **int counters[8]** — несколько счетчиков общего назначения;
- **void \*Link** — указатель общего назначения.

Очередные поля довольно важны и помогают анимировать и отслеживать состояние модели:

- **int anim\_state** — состояние анимации, которое в нашем контексте означает, какая именно анимация в данный момент проигрывается. Изначально в **id Software** был

разработан определенный список поддерживаемых анимаций, однако со временем многие приобрели лицензию на этот список и создали дополнительные анимации, так что в **конце** концов мне пришлось обратиться к программному коду *Quake II* и выяснить, сколько анимаций он поддерживает, как они называются и какими индексами кадров представлены. Мои поиски дали такой результат:

- в файле **.MD2** в общей сложности 198 кадров;
- в **нем** 20 различных анимаций;
- каждая анимация состоит из некоторого количества кадров, и это количество всегда одно и то же.

В табл. 15.1 приведен список анимаций с именами и индексами. Я составил этот список, проработав код *Quake II* к файл **M\_PLAYER.H** в каталоге заголовочных файлов исходного кода игры, который можно найти по адресу <http://www.fileplanet.com/files/80000/83000.shtml>.

Если эта ссылка вдруг окажется неработоспособной, можно воспользоваться поиском в Google или Yahoo.

**Таблица 15.1. Спецификации анимационных кадров формата .MD2**

<i>Номер анимации</i>	<i>Имя</i>	<i>Индексы кадров</i>
0	STANDING_IDLE	0-39
1	RUN	40-45
2	ATTACK	46-53
3	PAIN 1	54-57
4	PAIN 2	58-61
5	PAIN 3	62-65
6	JUMP	66-71
7	FLIP	72-83
8	SALUTE	84-94
9	TAUNT	95-111
10	WAVE	112-122
11	POINT	123-134
12	CROUCH STAND	135-153
13	CROUCH WALK	154-159
14	CROUCH ATTACK	160-168
15	CROUCH PAIN	169-172
16	CROUCH DEATH	173-177
17	DEATH BACK	178-183
18	DEATH FORWARD	184-189
19	DEATH SLOW	190-197

Большинство описаний дают представление о характере анимации. Например, PAIN 1, PAIN 2 и PAIN 3 — это анимации на случай, когда персонаж ударили и он "чувствует боль";

ATTACK — анимация, когда модель стреляет из своего оружия и т.д. Однако самые важные данные — это количество кадров: оно не меняется, оно всегда одно и то же. Проблема заключается в том, что многие из моделей, которые вы найдете в Internet, некорректны, и могут содержать либо слишком много, либо слишком мало кадров для отдельной анимации. Это не очень большая проблема, но рано или поздно вам придется с ней разобраться, потому что иначе у вас будут неприятности с анимацией. Лучшим решением будет использовать программируемые табличные или кадровые индексы, которые при необходимости можно изменять "на ходу" для каждой модели. Я создал группу определений, чтобы помочь выбрать `anim_state` в вашем коде.

```
// Состояния анимации md2
#define MD2_ANIM_STATE_STANDING_IDLE 0
    // Модель в состоянии покоя
#define MD2_ANIM_STATE_RUN 1
    // Модель бежит
#define MD2_ANIM_STATE_ATTACK 2
    // Стреляющая/атакующая модель
#define MD2_ANIM_STATE_PAIN_1 3
    // Модель получила удар версия 1
#define MD2_ANIM_STATE_PAIN_2 4
    // Модель получила удар версия 2
#define MD2_ANIM_STATE_PAIN_3 5
    // Модель получила удар версия 3
#define MD2_ANIM_STATE_JUMP 6
    // Прыгающая модель
#define MD2_ANIM_STATE_FLIP 7
    // Модель жестикулирует :)
#define MD2_ANIM_STATE_SALUTE 8
    // Модель салютует
#define MD2_ANIM_STATE_TAUNT 9
    // Модель дразнится
#define MD2_ANIM_STATE_WAVE 10
    // Модель угрожающе машет
#define MD2_ANIM_STATE_POINT 11
    // Модель прицеливается
#define MD2_ANIM_STATE_CROUCH_STAND 12
    // Модель, сгибаясь, замирает
#define MD2_ANIM_STATE_CROUCH_WALK 13
    // Модель идет наклонясь
#define MD2_ANIM_STATE_CROUCH_ATTACK 14
    // Модель стреляет из оружия, пригибаясь
#define MD2_ANIM_STATE_CROUCH_PAIN 15
    // Модель получает удар, будучи согнутой
#define MD2_ANIM_STATE_CROUCH_DEATH 16
    // Наклонившаяся модель умирает
#define MD2_ANIM_STATE_DEATH_BACK 17
    // Модель умирает, падая назад
#define MD2_ANIM_STATE_DEATH_FORWARD 18
    // Модель умирает, падая вперед
#define MD2_ANIM_STATE_DEATH_SLOW 19
    // Модель медленно умирает
```

Вы могли обратить внимание на отсутствие информации о частоте кадров. По существу, не мы, а наши устройства решают, как быстро проигрывать и интерполировать кадры; тем не менее, я добавил определенную поддержку для упрощения реализации анимации.

- `int anim_counter` — счетчик анимации, используемый для подсчета кадров, пока анимация обновляет следующий кадр.
- `int anim_mode` — определяет, как должна проигрываться анимация, задаваемая `anim_state`: разово или циклично. Обычно `anim_mode` принимает одно из приведенных значений.

```
// режим проигрывания анимации
#define MD2_ANIM_LOOP 0 // Проигрывать анимацию
                          // снова и снова
#define MD2_ANIM_SINGLE_SHOT 1 // Разовая анимация
```

- `int anim_speed` — скорость анимации, подсчитываемая при помощи `anim_counter` перед обновлением анимации. Чем меньше это число, тем быстрее проигрывается анимация.
- `int anim_complete` — этот флаг указывает, завершена ли анимация (1 — завершен, 0 — нет). Этот флаг можно использовать в логике игры, например, когда закончен прыжок персонажа, можете вывести соответствующий звук.
- `float curr_frame` — текущий визуализируемый кадр анимации. Заметим, что это число с плавающей точкой, а не целое. Дело в том, что поддержка интерполяции допускает дробные номера кадров. Например, кадр 5.4 означает интерполяцию между кадрами 5 и 6 на 40%. Мы подробнее остановимся на этом вопросе позже, когда будем рассматривать анимацию и интерполяцию вершин.

Давайте теперь напишем функцию загрузки. Итак, нам необходимо загрузить `.MD2`-файл с диска в буфер памяти, дописать заголовок, извлечь всю необходимую информацию и сохранить ее в классе-контейнере `MD2_CONTAINER`, после чего освободить временный буфер. Обычно я не привожу листинги таких громоздких функций, но в данном случае, я полагаю, это оправданное решение. Здесь приведена упрощенная версия функции из `T3DLIB13.CPP`, из которой удалена обработка ошибок; полную версию функции вы найдете на прилагаемом компакт-диске. До встречи в конце функции! :)

```
int Load_Object_MD2(MD2_CONTAINER_PTR obj_md2,
                   // Контейнер для модели .MD2
                   char *modelfile, // Имя файла модели .MD2
                   VECTOR4D_PTR scale, // Начальные коэффициенты
                                     // масштабирования
                   VECTOR4D_PTR pos, // Начальное положение
                   VECTOR4D_PTR rot, // Начальные повороты (не
                                     // реализованы)
                   char *texturefile, // Имя файла текстуры для модели
                   int attr, // Атрибуты освещения модели
                   int color, // Основной цвет при отсутствии
                             // текстуры
                   int vertex_flags) // Управляющие флаги
{
    // Функция загружает md2-файл, извлекает все данные и
    // сохраняет их в классе-контейнере, который будет
```

```

// использован позже для загрузки кадров в стандартный
// тип объекта для визуализации

FILE *fp = NULL;          // Файловый указатель модели
int flength = -1;         // Общий размер файла
UCHAR *buffer = NULL;     // Используется для буферизации
                           // данных md2 файла

MD2_HEADER_PTR md2_header; // Указатель на заголовок
                           // md2-файла

// Начинаем с загрузки модели
if ((fp = fopen(modelfile, "rb"))==NULL)
{
    Write_Error(
        "\nLoad_Object_MD2 — couldn't find file %s",
        modelfile);
    return(0);
} // if

// Определение размера файла
fseek(fp, 0, SEEK_END);
flength=fteU(fp);

// Считывание md2-файла в буфер для анализа

// Позиционирование в начале файла
fseek(fp, 0, SEEK_SET);

// Выделение памяти для файла
buffer = (UCHAR *)malloc(flength+1);

// Загрузка данных в буфер
int bytes_read = fread (buffer, sizeof(UCHAR),
                        flength, fp);

// Первым в буфере идет заголовок, так что для доступа к
// нему указатель на буфер можно просто приравнять
// указателю на заголовок
md2_header = (MD2_HEADER_PTR)buffer;

// Проверяем корректность формата
if (md2_header->identifier != MD2_MAGIC_NUM ||
    md2_header->version != MD2_VERSION)
{
    fclose(fp);
    return(0);
} // if

// Поля контейнера
obj_md2->state = 0;        // Состояние модели
obj_md2->attr = attr;      // Атрибуты модели
obj_md2->color = color;    // Основной цвет

```

```

obj_md2->num_frames = md2_header->num_frames;
    // Количество кадров в модели
obj_md2->num_polys = md2_header->num_polys;
    // Количество многоугольников
obj_md2->num_verts = md2_header->num_verts;
    // Количество вершин
obj_md2->num_textcoords = md2_header->num_textcoords;
    // Количество координат
    // текстур
obj_md2->curr_frame = 0;          // Текущий кадр анимации
obj_md2->skin = NULL;            // Указатель на текстуру
obj_md2->world_pos = *pos;       // Позиция объекта в мировых
                                // координатах

// Выделение памяти для сетки

// Указатель на список многоугольников
obj_md2->polys =
    (MD2_POLY_PTR)malloc(md2_header->num_polys *
        sizeof(MD2_POLY));
// Указатель на список координат вершин
obj_md2->vlist =
    (VECTOR3D_PTR)malloc(md2_header->num_frames *
        md2_header->num_verts * sizeof(VECTOR3D));
// Указатель на список координат текстур
obj_md2->tlist =
    (VECTOR2D_PTR)malloc(md2_header->num_textcoords *
        sizeof(VECTOR2D));

for (int tindex = 0; tindex < md2_header->num_textcoords;
    tindex++)
{
    // Вставка координат текстур в контейнер
    obj_md2->tlist[tindex].x =
        ((MD2_TEXTCOORD_PTR)(buffer +
            md2_header->offset_textcoords))[tindex].u;
    obj_md2->tlist[tindex].y =
        ((MD2_TEXTCOORD_PTR)(buffer +
            md2_header->offset_textcoords))[tindex].v;
} // for tindex

for (int findex = 0; findex < md2_header->num_frames;
    findex++)
{
    MD2_FRAME_PTR frame_ptr =
        (MD2_FRAME_PTR)(buffer +
            md2_header->offset_frames +
            md2_header->framesize * findex);

    // Извлечение коэффициентов масштабирования и
    // переноса и выполнение соответствующих операций
    float sx = frame_ptr->scale[0],
        sy = frame_ptr->scale[1],

```

```

        sz = frame_ptr->scale[2],
        tx = frame_ptr->translate[0],
        ty = frame_ptr->translate[1],
        tz = frame_ptr->translate[2];

for (int vindex = 0;
    vindex < md2_header->num_verts;
    vindex++)
{
    VECTOR3D v; // Временный вектор

    // Масштабирование и перенос
    v.x = (float)frame_ptr->vlist[vindex].v[0]*sx+tx;
    v.y = (float)frame_ptr->vlist[vindex].v[1]*sy+ty;
    v.z = (float)frame_ptr->vlist[vindex].v[2]*sz+tz;

    v.x = scale->x * v.x;
    v.y = scale->y * v.y;
    v.z = scale->z * v.z;

    float temp; // Используется для обмена

    // Внесение дополнительных изменений в
    // координаты вершин
    if (vertex_flags & VERTEX_FLAGS_INVERT_X)
        v.x = -v.x;
    if (vertex_flags & VERTEX_FLAGS_INVERT_Y)
        v.y = -v.y;
    if (vertex_flags & VERTEX_FLAGS_INVERT_Z)
        v.z = -v.z;
    if (vertex_flags & VERTEX_FLAGS_SWAP_YZ)
        SWAP(v.y, v.z, temp);
    if (vertex_flags & VERTEX_FLAGS_SWAP_XZ)
        SWAP(v.x, v.z, temp);
    if (vertex_flags & VERTEX_FLAGS_SWAP_XY)
        SWAP(v.x, v.y, temp);

    // Вставка вершины в список:
    // кадр 0, кадр 1,..., кадр n
    // кадр i: вершина 0, вершина 1,...,вершинаj
    obj_md2->vlist[vindex +
        (index * obj_md2->num_verts)] = v;

} // for vindex
} // for index

MD2_POLY_PTR poly_ptr = (MD2_POLY_PTR)(buffer +
    md2_header->offset_polys);

for (int pindex = 0; pindex < md2_header->num_polys;
    pindex++)
{
    // Вставка многоугольника в список в контейнере

```

```

if (vertex_flags &
    VERTEX_FLAGS_INVERT WINDING ORDER)
{
    // Изменение порядка вершин
    obj_md2->polys[pindex].vindex[0] =
        poly_ptr[pindex].vindex[2];
    obj_md2->polys[pindex].vindex[1] =
        poly_ptr[pindex].vindex[1];
    obj_md2->polys[pindex].vindex[2] =
        poly_ptr[pindex].vindex[0];

    // Координаты текстуры
    obj_md2->polys[pindex].tindex[0] =
        poly_ptr[pindex].tindex[2];
    obj_md2->polys[pindex].tindex[1] =
        poly_ptr[pindex].tindex[1];
    obj_md2->polys[pindex].tindex[2] =
        poly_ptr[pindex].tindex[0];
}
else
{
    obj_md2->polys[pindex].vindex[0] =
        poly_ptr[pindex].vindex[0];
    obj_md2->polys[pindex].vindex[1] =
        poly_ptr[pindex].vindex[1];
    obj_md2->polys[pindex].vindex[2] =
        poly_ptr[pindex].vindex[2];

    obj_md2->polys[pindex].tindex[0] =
        poly_ptr[pindex].tindex[0];
    obj_md2->polys[pindex].tindex[1] =
        poly_ptr[pindex].tindex[1];
    obj_md2->polys[pindex].tindex[2] =
        poly_ptr[pindex].tindex[2];
}
}

// Закрытие файла
fclose(fp);

////////////////////////////////////
// Загрузка текстуры из дискового файла
Load_Bitmap_File(&bitmap16bit, texturefile);

// Создание растрового изображения соответствующего
// размера и глубины цвета
obj_md2->skin =
    (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));

// Инициализация изображения
Create_Bitmap(obj_md2->skin, 0, 0,
    bitmap16bit.bitmapinfoheader.biWidth,
    bitmap16bit.bitmapinfoheader.biHeight,

```

```

        bitmap16bit.bitmapinfoheader.biBitCount);

// Загрузка изображения
Load_Image_Bitmap16(obj_md2->skin,
    &bitmap16bit,0,0,BITMAP_EXTRACT_MODE_ABS);

// Освобождение памяти
Unload_Bitmap_File(&bitmap16bit);
if (buffer)
    free(buffer);

// Возврат кода успешного завершения
return (1);
} // Load_Object_MD2

```

При вызове функции `Load_Object_MD2()` ей нужно передать указатель на контейнер для хранения **.MD2-модели**, наряду с именем дискового **.MD2-файла**, коэффициентами масштабирования, положения и вращения (в настоящее время реализованы только масштабирование и перенос), именем **.BMP-файла** с 16-битовой текстурой размером 256x256, атрибутами и основным цветом, и **наконец**, флагами для изменения порядка вершин, изменения системы координат и пр.

Если вы внимательно ознакомились с прототипом функции, то должны были заметить, что я **постарался** сделать загрузчик **.MD2-файла** по возможности похожим на другие загрузчики файлов. Функция начинает работу с поиска **.MD2-файла на диске**, после чего она открывает и загружает его в один большой буфер памяти. Затем начинается стадия анализа: функция сканирует заголовок файла и извлекает всю необходимую информацию. Затем функция вносит все необходимые данные в **класс-контейнер** в распакованном виде, и **инициализирует** все поля **класса-контейнера**.

Одним из интересных моментов является код работы с флагами `vertex_flags`, который позволяет выполнить определенные преобразования модели — изменение порядка вершин многоугольников и обмен осей. Как вы помните, мы используем левую систему координат с обходом вершин по часовой стрелке. Файлы **.MD2** используют правую систему с обходом против часовой стрелки. Чтобы исправить ситуацию, нам надо всего лишь поменять местами оси *x* и *y*. Маленькая деталь может обернуться большой головной болью, если не **решить** проблему при помощи простого флага. После того как модель загружена и извлечена в **MD2\_CONTAINER** класс, мы готовы к работе. Все, что от нас требуется, — это написать функцию, которая сможет извлечь единичный кадр из контейнера и вставить его в объект **OBJECT4DV2**. Кроме того, нам нужны функции поддержки для управления выбранными анимациями и синхронизацией.

## Анимация файлов **.MD2**

Анимация — это просто, если вы знаете формат **.MD2-файла**, а также извлекли и преобразовали данные в более приемлемый для нас формат данных — **MD2\_CONTAINER**. При вызове загрузчику передается контейнер.

```

// Некоторые рабочие векторы
static VECTOR4D vs = {4,4,4,1};
static VECTOR4D vp = {0,0,0,1};

// загрузка объекта

```

```

Load_Object_MD2(&obj_md2,      // Контейнер
"/md2/q2mdl-t/tris.MD2",      // Имя файла .MD2 модели
&vs,                          // Масштаб и положение
&vp,
NULL,
"/md2/q2mdl-t/blade.bmp",     // Файл текстуры модели
POLY4DV2_ATTR_RGB16 |         // Атрибуты
POLY4DV2_ATTR_SHADE_MODE_FLAT |
POLY4DV2_ATTR_SHADE_MODE_TEXTURE,
RGB16Bit(255,255,255),
VERTEX_FLAGS_SWAP_YZ);        // Управляющие флаги

```

#### НА ЗАМЕТКУ

Эта модель называется "Tekkaman Blade" и найти ее можно на узле Polycount. Это хороший образец качества моделей, размещенных там. Автор этой заслуживающей внимания модели ~ Майкл Меллор (Michael "Magarnigal" Mellor).

В результате у нас будет .MD2-модель, загруженная в объект MD2\_CONTAINER obj\_md2 из файла "/md2/q2mdl-t/tris.MD2" с текстурой из файла "/md2/q2mdl-t/blade.bmp".

Все, что нам нужно сделать, — это извлечь список вершин из заданного кадра и вставить его в список вершин OBJECT4DV2. Заметим, что список многоугольников, координат текстур и т.д. в OBJECT4DV2 никогда не изменяется, потому что это косвенные массивы, которые содержат индексы вершин, а не сами вершины. По сути, мы создаем "носитель" OBJECT4DV2 и заполняем его всем, за исключением вершин сетки.

Мы задаем атрибуты, цвет, список многоугольников, координаты текстур, текстурную карту и прочее, но не вставляем никакие вершины. Это и есть подготовка. Когда мы готовы визуализировать кадр, мы вызываем функцию, которая для данного кадра анимации извлекает вершины из контейнера MD2\_CONTAINER и затем вставляет их в "носитель" OBJECT4DV2. Затем мы просто передаем OBJECT4DV2 процессору визуализации.

### Подготовка объекта OBJECT4DV2 для модели .MD2

Начнем с функции, которая подготавливает OBJECT4DV2 для превращения в носитель для MD2\_CONTAINER. Ниже предоставлен ее код (без обработки ошибок для экономии места).

```

int Prepare_OBJECT4DV2_For_MD2(
    OBJECT4DV2_PTR obj,      // Целевой объект
    MD2_CONTAINER_PTR obj_md2) // md2-объект
{
    // Эта функция подготавливает OBJECT4DV2 для
    // использования в качестве "носителя" кадров из
    // md2-контейнера, распределяет необходимую память,
    // настраивает поля и выполняет все возможные
    // предвычисления

    // Очистка и инициализация объекта
    memset(obj, 0, sizeof(OBJECT4DV2));

    // Задание состояния как активного и видимого
    obj->state =
        OBJECT4DV2_STATE_ACTIVE | OBJECT4DV2_STATE_VISIBLE;

    // Некоторая информация об объекте
    obj->num_frames = 1; // Всегда равно 1
}

```

```

obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME |
        OBJECT4DV2_ATTR_TEXTURES;

obj->num_vertices = obj_md2->num_verts;
obj->num_polys = obj_md2->num_polys;
obj->texture = obj_md2->skin;

// Положение объекта
obj->world_pos = obj_md2->world_pos;

// Выделение памяти для вершин и многоугольников
if (!Init_OBJECT4DV2(obj,
        obj->num_vertices,
        obj->num_polys,
        obj->num_frames))
{
    Write_Error("\n(can't allocate memory).");
}

// Вычисление среднего и максимального радиуса с
// использованием вершин нулевого кадра. Это не очень
// точный метод, поскольку в процессе анимации объект
// может существенно изменяться

// Сброс значений
obj->avg_radius[0] = 0;
obj->max_radius[0] = 0;

// Цикл вычисления радиуса
for (int vindex = 0; vindex < obj_md2->num_verts;
    vindex++)
{
    // Обновление значений радиусов
    float dist_to_vertex =
        sqrt(obj_md2->vlist[vindex].x *
            obj_md2->vlist[vindex].x +
            obj_md2->vlist[vindex].y *
            obj_md2->vlist[vindex].y +
            obj_md2->vlist[vindex].z *
            obj_md2->vlist[vindex].z );

    obj->avg_radius[0] += dist_to_vertex;

    // Обновление максимального радиуса
    if (dist_to_vertex > obj->max_radius[0])
        obj->max_radius[0] = dist_to_vertex;
}

// Расчет среднего радиуса
obj->avg_radius[0] /= obj->num_vertices;

// Копирование списка координат текстур (всегда один и

```

```

// тот же)
for (int tindex = 0; tindex < obj_md2->num_textcoords;
    tindex++)
{
    // Координаты текстур
    obj->tlist[tindex].x = obj_md2->tlist[tindex].x;
    obj->tlist[tindex].y = obj_md2->tlist[tindex].y;
}

// Список индексов многоугольников (всегда один и тот
// же)
for (int pindex=0; pindex < obj_md2->num_polys;
    pindex++)
{
    // Индексы многоугольников
    obj->plist[pindex].vert[0] =
        obj_md2->polys[pindex].vindex[0];
    obj->plist[pindex].vert[1] =
        obj_md2->polys[pindex].vindex[1];
    obj->plist[pindex].vert[2] =
        obj_md2->polys[pindex].vindex[2];

    // Указатель на список вершин
    obj->plist[pindex].vlist = obj->vlist_local;

    // Атрибуты многоугольника
    obj->plist[pindex].attr = obj_md2->attr;

    // Цвет многоугольника
    obj->plist[pindex].color = obj_md2->color;

    // Применение текстуры к многоугольнику
    obj->plist[pindex].texture = obj_md2->skin;

    // Координаты текстуры
    obj->plist[pindex].text[0] =
        obj_md2->polys[pindex].tindex[0];
    obj->plist[pindex].text[1] =
        obj_md2->polys[pindex].tindex[1];
    obj->plist[pindex].text[2] =
        obj_md2->polys[pindex].tindex[2];

    // Атрибуты координат текстуры
    SET_BIT (obj->vlist_local [
        obj->plist[pindex].vert[0] ].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
    SET_BIT (obj->vlist_local [
        obj->plist[pindex].vert[1] ].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
    SET_BIT (obj->vlist_local [
        obj->plist[pindex].vert[2] ].attr,
        VERTEX4DTV1_ATTR_TEXTURE);
}

```

```

// Режим материала для эмуляции версии 1.0
SET_BIT(obj->plist[pindex].attr,
        POLY4DV2_ATTR_DISABLE_MATERIAL);

// Установка флага активности
obj->plist[pindex].state = POLY4DV2_STATE_ACTIVE;

// Установка списка вершин
obj->plist[pindex].vlist = obj->vlist_local;

// Список координат текстур
obj->plist[pindex].tlist = obj->tlist;

// Извлечение индексов вершин
int vindex_0 = obj_md2->polys[pindex].vindex[0];
int vindex_1 = obj_md2->polys[pindex].vindex[1];
int vindex_2 = obj_md2->polys[pindex].vindex[2];

// Вычисление нормали (помните, что вершины
// расположены по часовой стрелке, u=p0->p1,
// v=p0->p2, n=uxv)
VECTOR4D u, v, n;

u.x = obj_md2->vlist[vindex_1].x -
      obj_md2->vlist[vindex_0].x;
u.y = obj_md2->vlist[vindex_1].y -
      obj_md2->vlist[vindex_0].y;
u.z = obj_md2->vlist[vindex_1].z -
      obj_md2->vlist[vindex_0].z;
u.w = 1;
v.x = obj_md2->vlist[vindex_2].x -
      obj_md2->vlist[vindex_0].x;
v.y = obj_md2->vlist[vindex_2].y -
      obj_md2->vlist[vindex_0].y;
v.z = obj_md2->vlist[vindex_2].z -
      obj_md2->vlist[vindex_0].z;
v.w = 1;

// Расчет векторного произведения
VECTOR4D_Cross(&u, &v, &n);

// Вычисление длины нормали и сохранение ее в
// nlength
obj->plist[pindex].nlength = VECTOR4D_Length(&n);

} // for poly

// Код успешного завершения
return(1);

} // Prepare_OBJECT4DV2_For_MD2

```

Функция `Prepare_OBJECT4DV2_For_MD2()` принимает два параметра — указатель на "носитель" `OBJECT4DV2`, который получит информацию о сетке `MD2_CONTAINER`, и указатель на `MD2_CONTAINER`, содержащий `.MD2-модель`, которую мы хотим анимировать и визуализировать.

Функция `Prepare_OBJECT4DV2_For_MD2()` инициализирует объект `OBJECT4DV2` и затем копирует из класса-контейнера `MD2_CONTAINER` всю информацию, которая не меняется от кадра к кадру (не копируются только вершины). Этой функции не нужно быть очень быстрой, потому что она вызывается при инициализации игры и, следовательно, скорость не имеет большого значения. По сути, функция создает список многоугольников, текстур и выполняет все необходимые вычисления, связанные с объектом, такие как расчет длин нормалей поверхности, средний и максимальный радиусы.

**ВНИМАНИЕ**

Длина всех нормалей многоугольника вычисляется исходя из информации о кадре 0. При анимации модели предвычисленная длина нормалей многоугольников остается без изменений. В более точной модели требуется хранить длину нормали каждого многоугольника для всех множеств вершин; однако это означает существенное увеличение объема необходимой памяти — для хранения массива длиной 198 элементов. Особой же разницы в конечной картине заметно не будет. Добавим, что радиусы объекта рассчитываются также с использованием данных только из кадра 0.

## Извлечение кадров из контейнера `MD2_CONTAINER`

После подготовки `OBJECT4DV2` в качестве носителя информации о вершинах в `MD2_CONTAINER`, осталось очень мало работы. Просто выберите кадр от 0 до 198, выясните, где в списке вершин находятся нужные вершины, и затем скопируйте их в хранилище вершин в `OBJECT4DV2`, как если бы кадр все время оставался одним и тем же. Это делается с помощью следующего кода.

```
int Extract_MD2_Frame(OBJECT4DV2_PTR obj, // Целевой объект
                     MD2_CONTAINER_PTR obj_md2) // Объект md2
{
    // Функция извлекает единственный кадр анимации из
    // md2-контейнера и сохраняет его в контейнере
    // object4dv2, так что для работы мы можем использовать
    // имеющуюся у нас библиотеку

    // Проверка допустимости количества кадров
    int frame_0, // Начало интерполяции кадра
        frame_1; // Конец интерполяции кадра

    // Шаг 1: проверка — не интерполированный ли это кадр?
    float lvalue = obj_md2->curr_frame -
        (int)obj_md2->curr_frame;

    // Это целое значение?
    if (lvalue == 0.0)
    {
        // Кадр без интерполяции
        frame_0 = obj_md2->curr_frame;

        // Проверка на переполнение
        if (frame_0 >= obj_md2->num_frames)
            frame_0 = obj_md2->num_frames-1;
    }
}
```

```

// Копирование списка вершин для выбранного кадра;
// базовый индекс списка вершин:
// (obj_md2->num_verts * obj_md2->curr_frame)
int base_index = obj_md2->num_verts * frame_0;

for (int vindex = 0; vindex < obj_md2->num_verts;
    vindex++)
{
    // Копирование вершин
    obj->vlist_local[vindex].x =
        obj_md2->vlist[vindex + base_index].x;
    obj->vlist_local[vindex].y =
        obj_md2->vlist[vindex + base_index].y;
    obj->vlist_local[vindex].z =
        obj_md2->vlist[vindex + base_index].z;
    obj->vlist_local[vindex].w = 1;

    // Флаги атрибутов
    SET_BIT (obj->vlist_local[vindex].attr,
        VERTEX4DTV1_ATTR_POINT);
    SET_BIT (obj->vlist_local[vindex].attr,
        VERTEX4DTV1_ATTR_TEXTURE);

} // for vindex

} // if
else
{
    // Интерполяция между curr_frame и
    // curr_frame+1 на основе lvalue
    frame_0 = obj_md2->curr_frame;
    frame_1 = obj_md2->curr_frame+1;

    // Проверка на переполнение
    if (frame_0 >= obj_md2->num_frames)
        frame_0 = obj_md2->num_frames-1;

    if (frame_1 >= obj_md2->num_frames)
        frame_1 = obj_md2->num_frames-1;

    // Интерполяция списка вершин для выбранного
    // кадра(ов); базовый индекс
    // (obj_md2->num_verts * obj_md2->curr_frame)
    int base_index_0 = obj_md2->num_verts * frame_0;
    int base_index_1 = obj_md2->num_verts * frame_1;

    // Интерполяция вершин
    for (int vindex = 0; vindex < obj_md2->num_verts;
        vindex++)
    {
        obj->vlist_local[vindex].x =
            ((1-lvalue)*obj_md2->vlist[vindex+base_index_0].x+

```

```

        ivalue*obj_md2->vlist[vindex+base_index_1].x);
obj->vlist_local[vindex].y=
((1-ivalue)*obj_md2->vlist[vindex+base_index_0].y+
  ivalue*obj_md2->vlist[vindex+base_index_1].y);
obj->vlist_local[vindex].z=
((1-ivalue)*obj_md2->vlist[vindex+base_index_0].z+
  ivalue*obj_md2->vlist[vindex+base_index_1].z);
obj->vlist_local[vindex].w =1;

// Задание атрибутов
SET_BIT (obj->vlist_local[vindex].attr,
  VERTEX4DTV1_ATTR_POINT);
SET_BIT (obj->vlist_local[vindex].attr,
  VERTEX4DTV1_ATTR_TEXTURE);
} // for vindex

} // if

// Возврат кода успешного завершения
return(1);

} // Extract_MD2_Frame

Функция Extract_MD2_Frame() получает указатели на целевой "носитель" OBJECT4DV2 и контейнер-источник MD2_CONTAINER. Функция извлекает кадр curr_frame из MD2_CONTAINER. Функция осуществляет интерполяцию по ключевому кадру. Давайте поговорим об этом подробнее.

Обратите внимание на то, что функция Extract_MD2_Frame() начинается с оператора if.
// Это целое значение?
if (ivalue == 0.0)
{
    Это проверка наличия дробной части в номере текущего кадра. Если ее нет, кадр извлекается из списка вершин. Например, представим, что curr_frame равен 26. Список вершин из 26 кадра будет скопирован в носитель OBJECT4DV2 при помощи базового индекса, вычисляемого путем умножения числа вершин в одной модели num_verts на 26 (рис. 15.12). В случае же, когда curr_frame — величина не целая, нам необходима интерполяция между ключевыми кадрами.

    Взгляните на рис. 15.13. Вы видите на нем несколько ключевых кадров с шагающей фигуркой, составленной из отрезков. Давайте несколько сгладим эту анимацию. Мы можем рассчитать положение каждой вершины между кадрами i и i+1. Пока ничего сверхъестественного нет, и каждая вершина движется линейно, интерполяция будет выглядеть очень прилично. Такова скрытая философия .MD2-файлов. Всего анимационных кадров 198, и они используются для 20 разных анимаций. Может показаться, что это много, но посчитайте сами — при скорости 60 fps, с учетом того, что средняя длина анимации примерно равна 198/20=10 кадров, получается, что время воспроизведения анимации составляет всего около 0.16 с. За такое время трудно что-либо заметить!

    Чтобы замедлить анимацию, можно либо вставить задержки перед переключением кадров, либо использовать интерполяцию. Вставка задержек позволит замедлить анимацию, но при этом она будет выглядеть очень прерывистой. Гораздо лучше выполнить межкадровую интерполяцию, основанную на положениях вершин в каждом кадре. Этим и занят второй блок кода Extract_MD2_Frame().

```

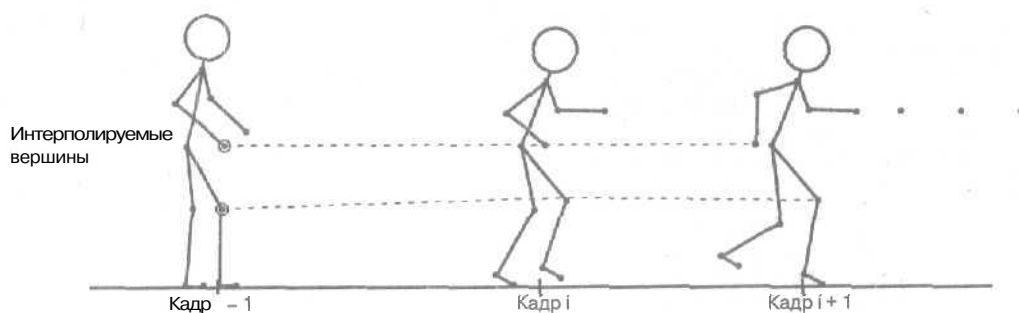


Рис. 15.13. Анимация по ключевым кадрам

## Реализация интерполяции кадров/вершин

Второй блок программного кода `Extract_MD2_Frame()` несколько сложнее первого в связи с тем, что именно он реализует описанную интерполяцию. Кадры для интерполяции выбираются на основании значения `curr_frame`. Если `curr_frame` не целое число, скажем, 5.1, то тогда нам надо интерполировать вершины из кадра 5 и кадра 6 так, чтобы получить 90% кадра 5 и 10% кадра 6.

Что касается математического описания данного процесса, то тут все просто. Пусть  $v_0$  — вершина из кадра 0, а  $v_1$  — это вершина из кадра 1, и мы хотим выполнить интерполяцию, основанную на значении `ivalue` из диапазона 0..1.0, которое представляет собой дробную часть номера текущего кадра `curr_frame`. Тогда  $v_i = v_0 \cdot (1 - \text{ivalue}) + v_1 \cdot \text{ivalue}$ .

Именно эти вычисления и выполняются упоминавшимся вторым блоком кода функции. Выделение частного случая целого значения `curr_frame` связано с тем, что в этом случае можно обойтись без вычислений, что позволяет ускорить работу функции.

Функции можно передать любой (в т.ч. нецелый) номер кадра из диапазона 0..198. Функция `Extract_MD2_Frame()` выполняет в случае необходимости интерполяцию и загружает вершины в носитель `OBJECT4DV2`.

## Добавление к поддержке .MD2анимации, основанной на состоянии

Итак, мы можем загружать .MD2-файл с диска, превращать его в более удобный в работе класс-контейнер `MD2_CONTAINER`, извлекать оттуда кадры с анимацией вершин, и сохранять их в носителе `OBJECT4DV2`. На этом можно было бы и остановиться — вызвав функцию визуализации и передав ей объект `OBJECT4DV2`. Однако вспомните о всех перечисленных в табл. 15.1 вариантах анимации! Было бы прекрасно, если бы мы могли вызвать функцию, которая позволила бы нам инициализировать разовую и циклическую анимации и управлять ими. Так мы могли бы заставить модель бегать, прыгать, стоять на месте и т.п., на основе логики игры, а не жестко программируя ее. Давайте напишем последний набор функций для решения этой задачи — вы увидите, что это действительно просто.

Нам нужны две вещи: простая структура данных для хранения анимации и пара функций для управления ею и обновления полей `MD2_CONTAINER`. Начнем с анимации, которая определяется начальным кадром, конечным кадром, частотой интерполяции и скоростью воспроизведения.

```
// Простая анимация
typedef struct MD2_ANIMATION_TYP
{
    int start_frame; // Начальный и конечный кадры
```

```

int end_frame;
float irate; // Частота интерполяции
int anim_speed; // Скорость анимации
} MD2_ANIMATION, *MD2_ANIMATION_PTR;

```

Поля `start_frame` и `end_frame` просто содержат индексы из табл. 15.1, а поля `irate` и `anim_speed` нуждаются в небольшом пояснении. `irate` — это частота интерполяции или, другими словами, сколько дополнительных кадров вставляется между ключевыми. Например, если `irate` равно 0.1, интерполяция от кадра 5 к кадру 6 будет выглядеть следующим образом.

5.0, 5.1, 5.2, 5.3 ... 5.9.6.0.

В большинстве случаев неплохо работает частота интерполяции 0.25 или 0.50. Старайтесь использовать для `irate` простые дроби. Следующее интересное поле — `anim_speed`. Это значение счетчика, используемого для выполнения анимации или логической схемы интерполяции по ходу игры, так что 0 означает, что в анимации участвуют все кадры, значение 1 — обновлять анимацию через кадр и так далее. Следующее, что нужно сделать, — это создать массив этих структур, каждая из которых содержит индексы анимации из табл. 15.1.

```

MD2_ANIMATION md2_animations[NUM_MD2_ANIMATIONS] =
{
    // Формат: начальный кадр (0..197),
    // конечный кадр (0..197),
    // коэффициент интерполяции (0..1, 1 — интерполяция
    // отсутствует), скорость (0..10, 0 - очень быстро,
    // 1 - быстро, 2 - средне, 3 - медленно...)

    { 0,39,0.5,1}, // MD2_ANIM_STATE_STANDING_IDLE 0
    {40,45,0.5,2}, // MD2_ANIM_STATE_RUN 1
    {46,53,0.5,1}, // MD2_ANIM_STATE_ATTACK 2
    {54,57,0.5,1}, // MD2_ANIM_STATE_PAIN_1 3
    {58,61,0.5,1}, // MD2_ANIM_STATE_PAIN_2 4
    {62,65,0.5,1}, // MD2_ANIM_STATE_PAIN_3 5
    {66,71,0.5,1}, // MD2_ANIM_STATE_JUMP 6
    {72,83,0.5,1}, // MD2_ANIM_STATE_FLIP 7
    {84,94,0.5,1}, // MD2_ANIM_STATE_SALUTE 8
    {95,111,0.5,1}, // MD2_ANIM_STATE_TAUNT 9
    {112,122,0.5,1}, // MD2_ANIM_STATE_WAVE 10
    {123,134,0.5,1}, // MD2_ANIM_STATE_POINT 11
    {135,153,0.5,1}, // MD2_ANIM_STATE_CROUCH_STAND 12
    {154,159,0.5,1}, // MD2_ANIM_STATE_CROUCH_WALK 13
    {160,168,0.5,1}, // MD2_ANIM_STATE_CROUCH_ATTACK 14
    {169,172,0.5,1}, // MD2_ANIM_STATE_CROUCH_PAIN 15
    {173,177,0.25,0}, // MD2_ANIM_STATE_CROUCH_DEATH 16
    {178,183,0.25,0}, // MD2_ANIM_STATE_DEATH_BACK 17
    {184,189,0.25,0}, // MD2_ANIM_STATE_DEATH_FORWARD 18
    {190,197,0.25,0}, // MD2_ANIM_STATE_DEATH_SLOW 19
};

```

В большинстве случаев я использую значение 0.5 для интерполяции и 1 — для обновления анимации. Имея заполненную структуру данных, нам не хватает только пары функций для запуска и воспроизведения анимации. Ниже представлена функция, которая создает отдельную анимацию.

```

int Set_Animation_MD2(
    MD2_CONTAINER_PTR md2_obj, // md2 объект

```

```

int anim_state, // Воспроизводимая анимация
int anim_mode = MD2_ANIM_LOOP) // Режим анимации
{
    // Инициализация воспроизведения анимации.
    md2_obj->anim_state = anim_state;
    md2_obj->anim_counter = 0;
    md2_obj->anim_speed =
        md2_animations[anim_state].anim_speed;
    md2_obj->anim_mode = anim_mode;

    // Установка начального кадра
    md2_obj->curr_frame =
        md2_animations[anim_state].start_frame;

    // Установка флага завершения анимации
    md2_obj->anim_complete = 0;

    // Возврат кода успешного завершения
    return(1);
} // Set_Animation_MD2

```

Функция `Set_Animation_MD2()` получает указатель на структуру `MD2_CONTAINER` модели, которую вы хотите анимировать, вместе с состоянием анимации и ее режимом. Предположим, например, что была загружена и подготовлена модель `obj_md2` типа `MD2_CONTAINER`. Для запуска анимации в циклическом режиме необходимо осуществить вызов `Set_Animation_MD2(&obj_md2, MD2_ANIM_STATE_RUN, MD2_ANIM_LOOP);`

Однако нам надо внести определенные изменения в анимацию и изменить поле `curr_frame` в классе `MD2_CONTAINER`, так что последняя функция анимации выглядит следующим образом.

```

int Animate_MD2(MD2_CONTAINER_PTR md2_obj) // md2 объект
{
    // Анимация каркаса — переход к следующему кадру

    // Обновление счетчика анимации
    if (++md2_obj->anim_counter >= md2_obj->anim_speed)
    {
        // Сброс счетчика
        md2_obj->anim_counter = 0;

        // Анимация с применением интерполяции
        // Если коэффициент интерполяции irate=1.0,
        // то интерполяция отсутствует

        // Добавление частоты интерполяции
        md2_obj->curr_frame +=
            md2_animations[md2_obj->anim_state].irate;

        // Проверка завершения последовательности
        if (md2_obj->curr_frame >
            md2_animations[md2_obj->anim_state].end_frame)
        {

```

```

// Проверка режима цикличности
if (md2_obj->anim_mode == MD2_ANIM_LOOP)
{
    // Цикл анимации к начальному кадру
    md2_obj->curr_frame =
        md2_animations [md2_obj->anim_state].
            start_frame;
}
else
{
    // MD2_ANIM_SINGLE_SHOT
    md2_obj->curr_frame =
        md2_animations [
            md2_obj->anim_state].end_frame;

    // Установка флага завершения
    md2_obj->anim_complete = 1;
}

} // if

} // if

// Возврат кода успешного завершения
return(1);

} // Animate_MD2

```

Вы просто вызываете функцию `Animate_MD2()`, передавая ей указатель на класс-контейнер, и она делает все необходимое. Учтите, что эта функция ничего не знает об интерполяции — она только обеспечивает синхронизацию и выполняет разовую или циклическую анимацию.

## Демонстрационные программы

Взгляните на копии экранов, показанные на рис. 15.14. Это результат работы демонстрационной программы `DEMO115_1.CPP|EXE`, которая осуществляет загрузку дискового `.MD2`-файла, а затем выполняет анимацию. На рисунке показаны два режима работы — объемный и каркасный. В программе используются следующие элементы управления:

- <1> — предыдущая анимация;
- <2> — следующая анимация;
- <3> — разовое воспроизведение анимации;
- <4> — циклическое воспроизведение анимации;
- <H> — экранная подсказка;
- клавиши управления курсором — перемещение камеры;
- <Esc> — выход.

Об остальных элементах управления рассказывается в экранной подсказке. Вы можете взять исходный текст программы и попробовать изменять величины `irate` и `anim_speed` в массиве `md2_animations[]`. Чтобы скомпилировать программу, вам понадобятся файлы `DEMO115_1.CPP|H`, а также все необходимые библиотечные файлы, включая файлы

T3DLIB1-13.CPP и библиотечные файлы DirectX. Само собой разумеется, на прилагаемом компакт-диске есть готовый скомпилированный .EXE-файл.

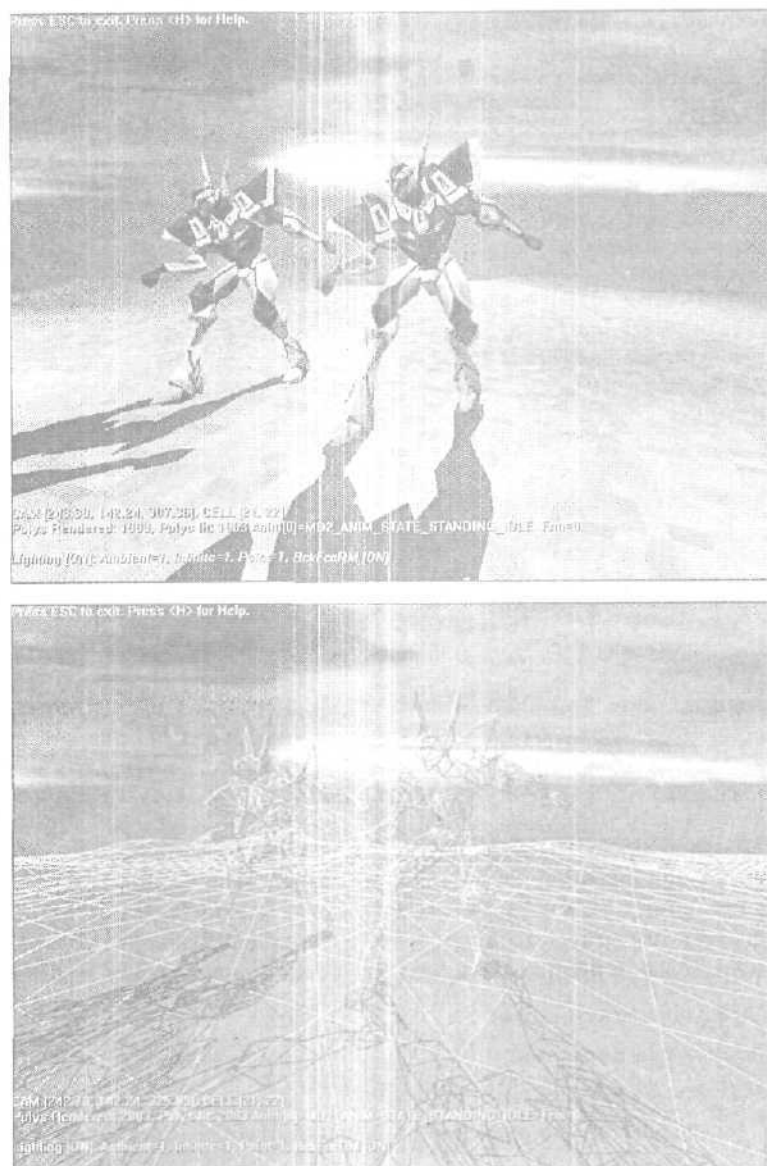


Рис. 15.14. Копии экрана демонстрационной программы для работы с .MD2-файлами

**ВНИМАНИЕ**

Эта демонстрационная программа использует файлы из вспомогательного подкаталога, так что при копировании программы с компакт-диска на жесткий диск не забудьте скопировать и подкаталоги со всем содержимым.

## Простая анимация без участия персонажа

Я хочу, чтобы вы поняли; в написании игр нет ничего сложного, надо только уметь выбрать правильный подход к решению той или иной задачи. Давайте рассмотрим, каким образом следует перемещать и анимировать объекты.

Конечно, мы уже знаем, как выполняется анимация персонажа из .MD2-файла, так что далее мы просто говорим об анимации точечных объектов. Вы же будете использовать технологии анимации персонажей наряду с рассматриваемыми далее идеями — с тем, чтобы персонажи выполняли определенные дополнительные действия при перемещении, поворотах и т.п.

## Вращательное и поступательное движение

Давайте начнем с двух основных типов анимации, которые вы можете применить к трехмерному объекту: вращение и перемещение. Вращательная анимация используется тогда, когда при перемещении объекта из одной точки в другую его надо повернуть (либо для стационарных объектов, которые должны поворачиваться — например, орудийной башни). Поступательное движение — это единственный способ переместить объект из точки A в точку B.

### Вращение объектов

Чтобы повернуть любой объект, нужно просто найти соответствующую матрицу поворота и потом применить ее к объекту. Мы неоднократно делали это в демонстрационных программах. Интересно здесь лишь создание высокоуровневой функциональности для управления вращением.

Здесь же надо вспомнить и о проблеме хранения преобразованных координат. Как уже говорилось, у всех моделей множество вершин в локальных координатах хранится в массиве `vlist[]`. Это единственная "свежая" копия модели. Когда мы преобразуем объект, мы можем либо выполнить трансформацию этих локальных координат, либо сделать их копию и сохранить результаты преобразования в массиве `tvlist[]`. Этот способ вполне работоспособен, однако результаты всех преобразований теряются при подготовке очередного кадра — объект просто "не помнит", что с ним было до сих пор.

Тем не менее, обычно этот способ наиболее приемлем, поскольку позволяет избежать деформации модели в связи с накоплением ошибок вычислений. Таким образом, лучше всего использовать некоторую переменную, отслеживающую текущие углы поворота объекта, а затем для каждого очередного кадра можно использовать эти данные для преобразования локальных координат с последующей визуализацией (рис. 15.15).



Рис. 15.15. Преобразование локальных координат и сохранение результатов в другом месте

Конечно, обратная сторона такого подхода заключается в том, что вы должны осуществлять преобразование вашей модели для каждого кадра. Предположим, на-

пример, что вы загружаете вашу модель и систему, где главная ось направлена в положительном направлении оси  $z$ . Для отслеживания поворота вокруг оси  $y$  понадобится введение дополнительной переменной (назовем ее `rot_y`). Ее надо будет постоянно обновлять и использовать при создании матрицы преобразования для каждого кадра. Таким образом, если в вашей модели содержится 1000 вершин, то для каждого кадра вам придется совершать 1000 преобразований координат вершин из локальных в мировые. Это — плата за то, что при любых преобразованиях модель не окажется деформирована.

#### СОВЕТ

Если вы не можете выполнять все необходимые преобразования для каждого кадра, то можно пойти на преобразование координат модели с сохранением результатов преобразования. При этом можно, например, каждые 10000 кадров "освежать" локальные координаты модели при помощи копии исходной модели, чтобы минимизировать деформацию из-за ошибок вычислений.

Я надеюсь, что вы испытаете оба подхода и сами увидите, какой из них в большей степени подходит для вас. Думаю также, что вам определенно стоит потратить время и на создание кода, с помощью которого вы сможете заставить объект поворачиваться на определенный угол за определенное количество кадров.

## Линейное перемещение объектов

Перемещение объекта по прямой — задача тривиальная; это всего лишь обычный перенос. Однако существует множество способов его осуществления. Один из них заключается в использовании простого вектора скорости  $V$ , который добавляется к радиус-вектору  $r$  для получения нового положения объекта в следующем кадре:  $r = r + v$ . Так выполняется единичное перемещение объекта со скоростью  $t$ . А что, если вы хотите переместить объект из точки  $p_0$  в точку  $p_1$ ? На самом деле это легко: мы просто создаем параметрическое векторное уравнение:

$$R = p_0 + t \cdot (p_1 - p_0), \text{ где } t \in [0, 1].$$

Как видите, радиус-вектор  $r$  прочертит прямолинейный отрезок из точки  $p_0$  в точку  $p_1$  при изменении значения параметра  $t$  от 0 до 1.0. Скорость перемещения управляется величиной изменения параметра за единицу времени (рис. 15.16).

## Сложное параметрическое и криволинейное движение

Хотя прямые линии элегантны с точки зрения геометрии, они слишком скучны. Следующим шагом должно стать перемещение объектов по криволинейным траекториям (т.е. по более сложным параметрическим кривым).

#### НА ЗАМЕТКУ

Движение может быть реализовано при помощи физических моделей реального времени — результат окажется очень реалистичным, основанным на вычислениях с учетом массы объекта и полей сил. Здесь же мы рассматриваем только простейшее детерминистическое движение практически без учета законов физики.

Создавать криволинейные траектории так же просто, как и прямолинейные; просто надо использовать более сложные параметрические уравнения. Скажем, вы хотите переместить объект вверх по спирали, как показано на рис. 15.17. Для решения этой задачи достаточно использовать формулу для окружности и создать ее параметрическую версию для перемещения радиус-векторар.

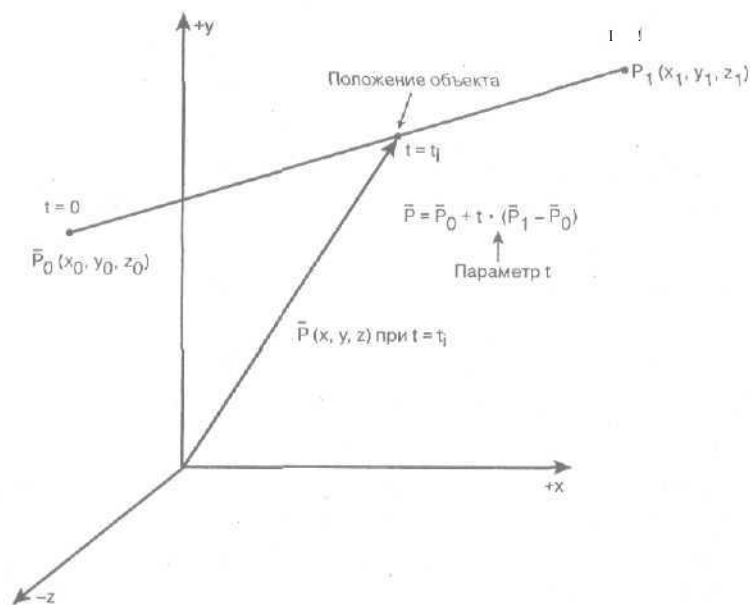


Рис. 15.16. Параметрическое движение по прямой линии

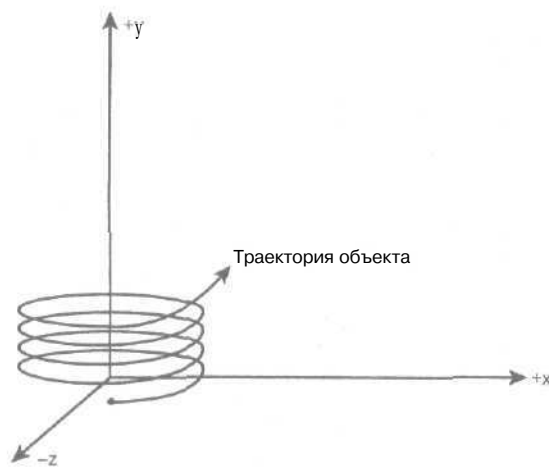


Рис. 15.17. Траектория движения по спирали

Так, для перемещения точки по окружности или эллипсу в плоскости  $x$ - $z$  можно воспользоваться следующими параметрическими уравнениями:

$$p_x = p_{0x} + r_1 \cdot \cos(2\pi c t),$$

$$p_z = p_{0z} + r_2 \cdot \sin(2\pi c t).$$

Если добавить к этому уравнение для движения с постоянной скоростью вдоль оси  $y$

$$p_y = p_{0y} + b \cdot t,$$

то мы получим полное решение поставленной задачи движения по спирали. Здесь  $r_1$  и  $r_2$  — радиусы эллипса вдоль осей  $x$  и  $z$ ,  $c$  — константа, определяющая скорость вращения,  $b$  — константа, определяющая скорость подъема вдоль оси  $y$ ,  $at$  — параметр.

Это все, что нужно для реализации сложных криволинейных траекторий. Большинство из них можно создать при помощи синусоидальных кривых, но это отнюдь не ограничение, и вы можете строить и более сложные траектории, используя полиномиальные кривые, кривые Безье или сплайны. Если помните, то при рассмотрении аксонометрических преобразований мы использовали квадратичные кривые для интерполяции пути между точками. Вы можете использовать аналогичный подход при интерполяции кривых движения, создавая параметрическую модель кривой в координатах  $x$ ,  $y$ ,  $z$  и вычисляя точки траектории объекта. Конечно, вам придется "сшить" в одно целое множество сегментов кривой, но это не так уж и трудно.

#### НА ЗАМЕТКУ

Когда объект находится в трехмерном пространстве, мы имеем дело не только с его перемещением вдоль траектории, но и с его вращением по ходу движения. Т.е. мы можем при желании заставить, например, космического воина поворачиваться по мере его перемещения. Это несложно — надо только добавить вращение параллельно вектору линейного перемещения от предыдущего положения к следующему, или параллельно касательной к кривой в текущем положении объекта.

## Использование сценариев движения

Последний прием создания движения, о котором я хочу рассказать, — это сценарии. Суть заключается в создании небольшой программы, которая будет управлять движением и анимацией персонажа игры. Главная идея состоит в том, чтобы создать множество маленьких сценариев для ваших персонажей; соответствующий процессор анализирует их команды и определяет действия персонажей. Данная концепция проиллюстрирована на рис. 15.18.

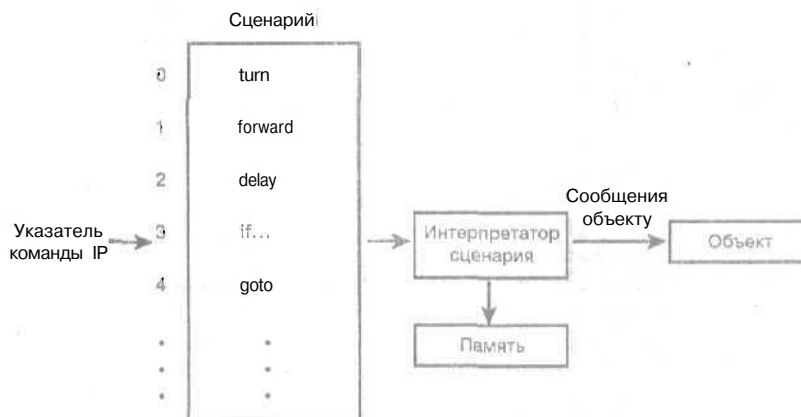


Рис. 15.18. Архитектура процессора сценариев

Используя сценарии, вы можете создавать действительно сложные модели поведения, заставляя персонажей игры передвигаться по замысловатым траекториям, что практически недостижимо с использованием жестко закодированной логики или путей. Кроме того, вы можете создавать собственные высокоуровневые языки для написания сценариев, которые позволят вам самостоятельно программировать персонажи.

#### НА ЗАМЕТКУ

Если вы хотите больше узнать о создании сценариев в играх, ознакомьтесь с лучшей книгой по данному вопросу — Varanese A., Romero J. *Game Scripting Mastery*. Premier Press.

В качестве примера давайте определим язык, поддерживающий прямолинейное движение и вращение вокруг любой оси с поддержкой задержек. Мы создадим маленький сценарий в виде массива или списка команд, а затем напишем анализатор, который читает и выполняет команды сценария. Ниже показано, как может выглядеть возможная структура данных, используемая в сценарии.

```
// Формат инструкции
typedef struct OPCODE_TYP
{
    int opcode;      // Код команды
    float op1;       // Операнд (число с плавающей точкой)
    VECTOR3D op2;    // Векторный операнд
} OPCODE, *OPCODE_PTR;
```

Каждая команда имеет код и два (необязательных) операнда. Вот некоторые из кодов.

```
#define OP_MOVE 0 // Движение
#define OP_ROTATE_X 1 // Поворот вокруг оси x
#define OP_ROTATE_Y 2 // Поворот вокруг оси y
#define OP_ROTATE_Z 3 // Поворот вокруг оси z
#define OP_DELAY 4 // Время выдержки
#define OP_END -1 // Завершение программы
```

Для каждого кода необходимо определить используемые операнды, например, следующие.

- OP\_MOVE — новое положение объекта сохраняется в op2, а скорость — в op1 (например, 0 — очень медленно, а 1 — очень быстро);
- OP\_ROTATE\_\* — для всех кодов вращения угол поворота в градусах записывается в op1. Поддерживаются как положительные, так и отрицательные значения угла;
- OP\_DELAY — число кадров задержки хранится в op1; значение должно быть целым;
- OP\_END — безусловное прекращение выполнения сценария.

Все очень просто. Дело за малым — реализацией процессора в коде. Ниже представлен возможный вариант.

```
int instruction_ptr = 0; // Указатель команды
int fetch = 1;          // Используется для выборки
                        // команды
OPCODE instr;           // Следующая команда

while(1)
{
    // Выборка очередной команды
    if (fetch)
    {
        instr = pattern[instruction_ptr];
        fetch = 0;
    } // if

    // Выбор кода
    switch(instr.opcode)
    {
        case OP_MOVE: // Код перемещения
        {
```

```

        // Перемещаем объект
    } break;

    case OP_ROTATE_X: // Поворот вокруг оси x
    {
        // Поворот вокруг оси x на угол
        // instr.op1
    } break;
    case OP_ROTATE_Y: // Поворот вокруг оси y
    {
        // Поворот вокруг оси y на угол
        // instr.op1
    } break;
    case OP_ROTATE_Z: // Поворот вокруг оси z
    {
        // Поворот вокруг оси z на угол
        // instr.op1
    } break;
    case OP_DELAY: // Задержка
    {
        // Отсчет instr.op1 кадров
    } break;
    case OP_END: // Завершение сценария
    {
        // Завершить цикл, программу и
        // т.д.
    } break;
    default: break;
} // switch

} // while

```

Конечно, после выполнения каждого действия флаг `fetch` устанавливается равным 1, так что происходит выборка очередной инструкции. В дополнение к коду, нам нужен сценарий, и ниже представлен его образец.

```

OPCODE program[] = {
    {OP_MOVE, 1, 100, 0, 0},
    {OP_MOVE, 1, 100, 100, 0},
    {OP_MOVE, 1, 0, 100, 0},
    {OP_MOVE, 1, 100, 0, 0},
    {OP_DELAY, 100, 0, 0, 0},
};

```

Хотя этот пример чрезвычайно упрощен, основная идея **понятна** — использование высокоуровневых **сценариев** для управления движением персонажей на низком уровне.

## Обнаружение трехмерных столкновений

Обнаружение столкновений — не что иное, как геометрическая проблема самого низкого уровня, и существует масса интересных алгоритмов обнаружения пересечений или соударения двух объектов. Что я хочу реально сделать — так это поощрить вас в желании упрощать: в 99% случаев простые ограничивающие сферы, боксы и т.п. работают не хуже действительно сложного алгоритма столкновения, так что давайте рассмотрим некоторые упрощенные подходы.

## Ограничивающие сферы и цилиндры

Первый метод обнаружения столкновений состоит в использовании ограничивающих сфер или цилиндров вокруг каждого объекта с последующими проверками столкновений этих ограничивающих объемов. Представим, что у нас есть два объекта А и В (рис. 15.19).

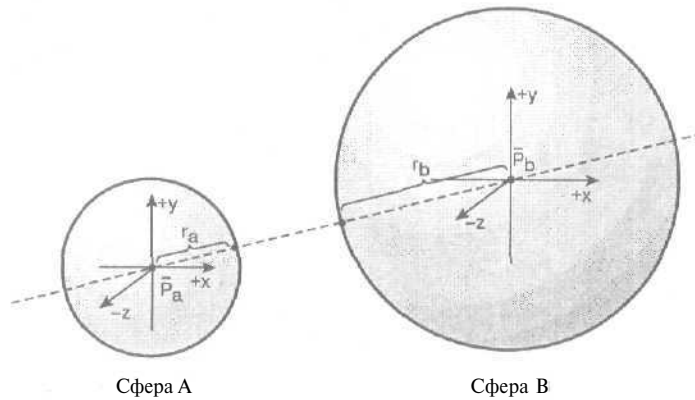


Рис. 15.19. Метод обнаружения столкновения ограничивающих сфер

Каждый объект окружен ограничивающей сферой с радиусами  $r_a$  и  $r_b$  для определения, имело ли место столкновение. Мы просто выясняем, не является ли расстояние между двумя центрами меньшим, чем  $(r_a + r_b)$ .

```
if (dist_xyz(p_a, p_b) < (r_a + r_b))  
{  
    // Произошло столкновение  
}
```

Здесь  $\text{dist}_{xyz}(p_a, p_b)$  — расстояние между точками  $p_a$  и  $p_b$  в трехмерном пространстве.

Конечно, если объекты в действительности вовсе не сферические, например, длинная труба или доска, применение ограничивающих сфер не слишком оправдывает ваши надежды. Однако по своему опыту написания игр я знаю, что можно решить проблему, используя "сжатые" ограничивающие сферы. Другими словами, вы уменьшаете ограничивающие сферы до 50–70% от их действительного размера.

К тому же можно использовать и другие ограничивающие формы, например, для удлиненных объектов — цилиндры. Если только цилиндры ориентированы одинаково, обнаружение столкновения становится столь же тривиальным (рис. 15.20).

На рисунке мы видим два объекта А и В с радиусами  $r_a$  и  $r_b$ , соответственно. Чтобы обнаружить их столкновение, нам необходимо проверить две вещи: не перекрываются ли объекты друг с другом по оси  $y$ , и не перекрываются ли их сечения.

```
if (dist_xz(p_a, p_b) < (r_a + r_b) &&  
    (fabs(p_a.y - p_b.y) < (h_1 + h_2)/2))  
{  
    // Произошло столкновение  
}
```

Здесь функция  $\text{dist}_{xz}(p_a, p_b)$  вычисляет расстояние между точками  $p_a$  и  $p_b$  в плоскости  $xz$ .

Конечно, для оптимального решения конкретной задачи можно подбирать различные ограничивающие объемы, например, ограничивающие боксы. Обнаружение столкнове-

ний выполняется аналогично — вы просто проверяете, не перекрываются ли данные геометрические объекты.

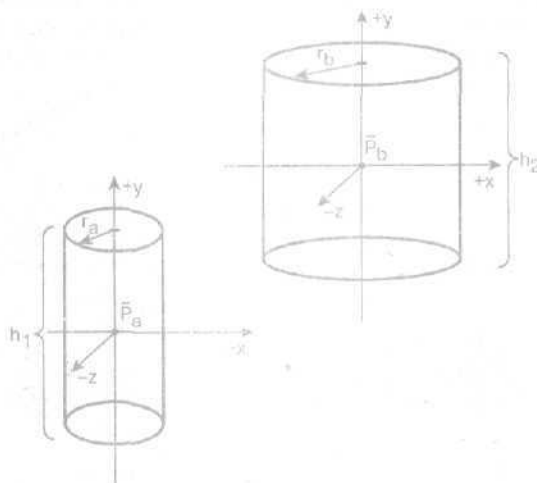


Рис. 15.20. Метод ограничивающих цилиндров для обнаружения столкновений

НА ЗАМЕТКУ

Рекомендую вашему вниманию книгу, посвященную обнаружению столкновений: Schneider Ph., Eberly D, *Geometric Tools for Computer Graphics*.

## Использование структур данных для ускорения обнаружения столкновений

Последнее, на чем я хотел бы остановиться в этой главе, — с обнаружением столкновений, — это оптимизация. Методы обнаружения столкновений подобны сложным математическим формулам: мы все их знаем, о них написано много книг, но для реальных вычислений они мало применимы. Между игрой в реальном времени и теоретическим определением столкновений есть одна разница — это скорость.

Например, представьте, что у вас есть 1000 объектов в игровом пространстве. А теперь предположим, что вы хотите обнаружить столкновение объектов. Итого надо провести  $1000 \cdot 999 / 2 = 499500$  проверок. Теперь представьте, что все эти объекты стреляют, их части разлетаются, и число объектов увеличивается до 10000. Теперь уже надо провести 49995000 вычислений! Я веду к тому, что на проблему обнаружения столкновений пора взглянуть и под другим углом, например, с точки зрения пространственного разделения.

Мы обсуждали пространственное разделение, когда рассматривали BSP-деревья, и я упоминал, что эти структуры данных полезны не только для графики, но и для обнаружения столкновений, и вы сами видите почему. Используя эти структуры, мы можем распределить наши объекты по ячейкам пространства таким образом, что в каждой такой части будет находиться не более пары сотен объектов. Количество расчетов при этом кардинально сократится.

Например, пусть у нас есть игра с объектами в количестве 10000, и пусть после деления игрового пространства мы получим в каждой части не более 50 объектов. Таким образом, нам надо будет обработать  $10000 / 50 = 200$  частей, в каждой из которых вы-

полняется не более  $50 \cdot 49/2 = 1225$  проверок, что в общей сложности дает 245000 проверок вместо 49995 000.

Это пример нестандартного мышления, полезного при разработке обнаружения столкновений. Не закидывайтесь на низкоуровневых алгоритмах, которые выполняют проверку пересечения ограничивающих боксов или многоугольников — начните с общего алгоритма столкновений, и вы уменьшите объем работы во много раз.

## Техника следования по поверхности

Это последняя тема, которую я хочу обсудить и по которой я получал много вопросов. Когда вы создаете игру с любым типом поверхности, вы должны следовать по ней вместе со всеми своими танками, персонажами и всем остальным движимым имуществом :). Для большинства поверхностей в демонстрационных программах во второй половине этой книги мы использовали очень простую, но эффективную физическую модель, которая сканирует многоугольную сетку по центру движущегося объекта, находит высоту вершин, создавая небольшой клочок земли, и затем поднимает или опускает объект пропорционально общему градиенту изменения расстояния от текущего положения к новому.

Это отлично смотрится и требует всего 10 строк кода. Однако проблема в том, что объект все еще не следует по поверхности. Иначе говоря, если у объекта есть, например, колеса, он будет подниматься и снижаться корректно, но при этом не следуя по поверхности. Давайте поговорим о том, как это исправить.

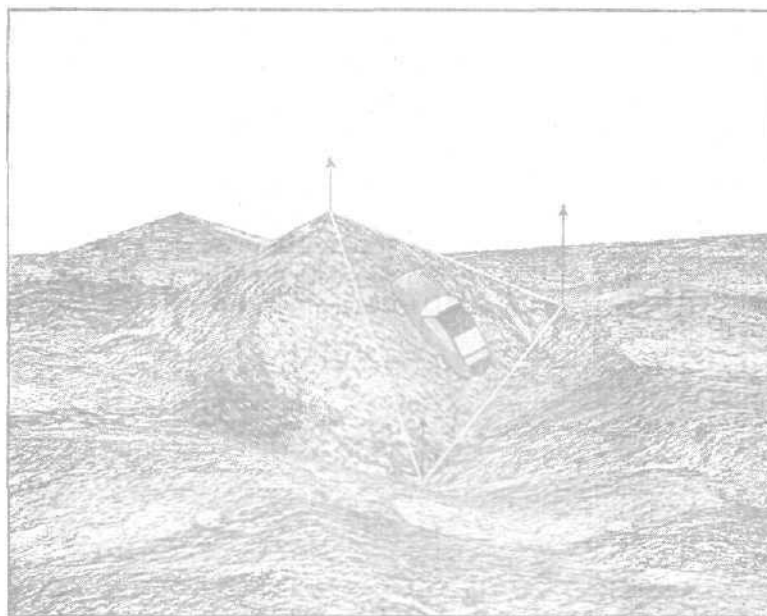


Рис. 15.21. Решение проблемы следования по поверхности

На рис. 15.21 показан кусочек поверхности, по которой мы хотим следовать. Вот один из возможных алгоритмов.

1. Текущая высота или положение объекта, который следует по поверхности, рассчитывается как средняя высота вершин участка поверхности, или, если вы стреми-

тес к большей точности, вы должны провести линию из центра объекта и посчитать, где она пересекает плоскость многоугольника участка, над которым "парит" объект. Конечно, если вы действительно хотите точности, вам придется таким же образом спроецировать и колеса или другие точки контакта объекта. В любом случае вы используете результат вычислений для поднятия или снижения объекта.

2. Следующий шаг — это вращательная ориентация объекта. Есть несколько путей решения данной проблемы, основанных на точках контакта. Однако самый быстрый (и грубый) метод таков: вычислить вектор нормали к участку поверхности. Затем повернуть объект так, чтобы его вектор нормали выровнялся с вектором нормали поверхности. Для этого нужно просто просчитать углы между нормалью поверхности и нормалью объекта. Здесь можно использовать, например, скалярное произведение, но есть и другие способы. После определения углов вращения вы должны создать соответствующую матрицу и преобразовать объект, выравняв его по нормали к поверхности.

Я опускаю подробности, но основные шаги — это перенести объект на подходящую высоту и повернуть его так, чтобы нормаль объекта стала параллельна нормали к участку поверхности.

## Резюме

В этой главе мы в конце концов научились анимировать трехмерные модели персонажей, и можем посмотреть, как они двигаются. В предыдущих главах мы рассмотрели огромное количество материала о движении и анимации, но возврат к уже пройденному только закрепляет полученные знания. Наконец, мы рассмотрели несколько основных схем обнаружения трехмерных столкновений и обсудили кое-какие идеи для более продвинутых систем.

Сейчас вы более чем хорошо подготовлены к самостоятельной разработке всех этих вещей, поскольку владеете основами трехмерной графики и растеризации. Это один из наиважнейших моментов. В следующей (и последней) главе мы рассмотрим несколько приемов оптимизации, которые можно использовать в нашем процессоре, и новые процессоры Pentium III+ с поддержкой SIMD (одна инструкция — множество данных).

# ГЛАВА 16

## Технологии оптимизации

### В этой главе...

• Введение в оптимизацию	1360
• Профилирование кода программы	1360
• Использование компилятора Intel C++	1372
• Введение в SIMD	1377
• Класс трехмерных векторов с поддержкой SIMD	1392
• Некоторые оптимизационные приемы	1400

На мой взгляд, скорость работы программы никогда не бывает достаточной, и в данной главе мы рассмотрим некоторые технологии оптимизации, позволяющие ее повысить. Мы не будем заниматься методами низкоуровневой оптимизации, поскольку их реализация требует слишком много времени. Вместо этого сконцентрируем наши усилия на использовании преимуществ новых технологий и программных средств, применяемых на этапе оптимизации при создании игры. Ниже перечислены основные темы, обсуждаемые в данной главе:

- введение в технологии оптимизации;
- профилирование кода с помощью Visual C++ и Intel VTune;
- использование Intel C++;
- введение в SIMD-программирование;
- некоторые приемы оптимизации.

## Введение в оптимизацию

Изначально я собирался предложить читателям стандартный набор: показать циклы, примеры программного кода и т.д., а затем показать, как они оптимизируются. Однако это уже неоднократно делалось как мною, так и другими авторами. В то же время я не уверен, что данная информация в *настоящее* время имеет такое же *существенное* значение, как та, которую я и буду излагать далее. Она действительно нетривиальна, и данная глава обогатит вас подлинно новыми знаниями. Поэтому я решил сосредоточиться на технологии, а не на *частностях*, и задал себе вопрос: какие средства я использую при оптимизации кода? Затем я провел небольшой опрос и, к своему *удивлению*, обнаружил, что многие разработчики игровых программ не только никогда не использовали эти средства и технологии, но часто даже никогда не слышали о них! Я был счастлив: наконец мне удалось найти нечто новое для большинства, следовательно, рассмотрение этого *вопроса* принесет несомненную пользу.

Ниже предлагается несколько вопросов. Если вы ответите отрицательно хотя бы на некоторые из них, предложенная в данной главе информация будет вам полезна.

- **Вопрос 1.** Приходилось ли вам использовать встроенные средства профилирования Microsoft Visual C++?
- **Вопрос 2.** Приходилось ли вам использовать инструмент оптимизации *VTune* компании Intel?
- **Вопрос 3.** Приходилось ли вам устанавливать и использовать компилятор Optimizing C++ компании Intel, подключаемый к Microsoft Visual C++?
- **Вопрос 4.** Приходилось ли вам программировать код SSE или SSE2? Эти сокращения обозначают Streaming Single Instruction Multiple Data Extensions (Потоковые расширения архитектуры с одним потоком команд и несколькими потоками данных) версии 1 и 2 для Pentium III и Pentium 4, соответственно (ошибки здесь нет, именно так компания Intel *обозначает* версии процессоров Pentium: III и 4).
- **Вопрос 5.** Читали ли вы руководства по архитектуре и программному *обеспечению* Pentium III/4, которые вполне *доступны* бесплатно?

Опросив примерно 20 разработчиков игр, я обнаружил, что 90% любителей ответили *отрицательно* практически на все эти вопросы. Я этого и ожидал, но меня поразило то, что 50% профессиональных разработчиков игр также ответили отрицательно на большую часть вопросов. Это означает, что большинство *разработчиков* не только не применяют существующие средства *оптимизации*, но и *процессоры Pentium III/4* работают у них в медленном режиме, не используя преимуществ суперскалярной архитектуры, *предоставляемых* SIMD и средствами параллельной обработки. Учитывая сказанное, я вижу *цель* данной главы в том, чтобы показать, насколько *просто* опробовать данные средства и технологии. В результате перед вами откроются совершенно новые возможности.

Наконец, в конце главы я продемонстрирую некоторые "хитрые" приемы.

## Профилирование кода программы

В старые добрые времена можно было выйти из положения, помещая в начале функции *вызов* функции замера времени и определяя время ее выполнения путем многократных (тысячи раз) вызовов. Можно было *эмпирически* исследовать программу; попробовать некую оптимизацию, затем *выполнить* повторную компиляцию и вновь запустить программу. Эти методы по-прежнему актуальны и для небольших программ с их *помощью* можно по-

лучить неплохие результаты. Однако при наличии современных **процессоров**, оптимизирующих компиляторов и систем программного обеспечения, содержащих наряду с вашим собственным кодом обращения к системе, динамически подключаемым библиотекам и интерфейсам прикладного **программирования**, задача определения, с чего начинать оптимизацию, может оказаться слишком утомительной и практически невыполнимой.

Предположим, вы пишете **приложение-растеризатор**. Можно оптимизировать внутренний цикл растеризатора, полагая, что главная работа производится здесь. Но при этом не принимается в расчет, что перед кодом **растеризатора** производится вызов функции времени исполнения `C тетсру()`, выполняющей побайтовое **перемещение**, которое можно **ускорить** в 4 раза. Как можно что-то оптимизировать, если вы не догадываетесь, что оно выполняется медленно? Необходимы некоторые средства, которые помогут построить таблицы и графики времени работы отдельных функций, использования памяти и прочих факторов, чтобы в идеале можно было посмотреть на "профиль" программы в целом и определить, на что следует потратить время. В этом и состоит цель профилирования программы.

Существует множество инструментов профилирования. Мы обсудим два из них, причем одно, вероятно, уже есть в вашем распоряжении, а второе можно получить бесплатно (по крайней мере, демонстрационную версию).

## Профилирование с помощью Visual C++

В версиях Microsoft Visual C++ Professional и Enterprise есть встроенные средства профилирования (в стандартной и образовательной версиях таких средств, увы, нет (.NET также имеет аналогичные возможности профилирования). Профилирование в Visual C++ осуществляется достаточно просто и сразу же дает вполне приличные результаты, которые позволят приступить к оптимизационному анализу и найти **фрагменты, подлежащие** перделке. В качестве примера профилируемой программы мы будем использовать **DEMOII16\_1.CPP|EXE**. Эта программа в основном идентична демонстрационной программе с персонажами из предыдущей главы, за исключением того, что она выполняется в оконном режиме, что позволяет производить с **ней** разные манипуляции и в случае ее сбоя не утратить контроль над компьютером. Первым делом необходимо скомпилировать программу.

Чтобы скомпилировать **DEMOII16\_1.CPP|EXE**, необходимо создать проект или использовать уже имеющийся. Это не должно вызвать затруднений. Кроме **того**, необходимо подключить модули **T3DLIB1-13.CPP**, а также библиотечные файлы **DirectX**, как это делалось в предыдущей главе при компиляции **соответствующей** демонстрационной программы. Когда программа будет надлежащим образом скомпилирована, при ее выполнении в окне появятся персонажи, как показано на рис. 16.1.

Теперь, когда у нас есть работающая демонстрационная программа и возможность ее компиляции, установим компилятор в режим профилирования. Для этого необходимо выполнить следующие действия.

1. В главном меню Visual C++ выберите **Project⇒Settings**. В листе **свойств** выберите **Link**, а затем в поле **Category** — **General**. В результате на экране появится окно, показанное на рис. 16.2. После этого необходимо установить флажки **Enable Profiling** и **Generate Debug Info**, как показано на рисунке, после чего щелкнуть на кнопке **ОК** и закрыть диалоговое окно.
2. Теперь необходимо собрать приложение с новыми настройками. Чтобы полностью перестроить приложение с возможностью профилирования, следует выбрать в главном меню **Build⇒Rebuild All**.



тически, но если это не произойдет, нужно щелкнуть мышью на данном ярлыке, как показано на рис. 16.4. Измените размер окна, чтобы можно было увидеть больше информации (рис. 16.5). Теперь проанализируем то, что мы видим.

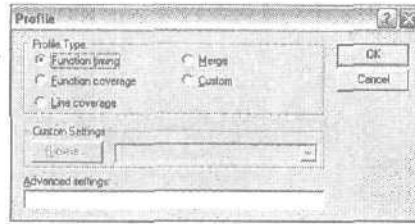


Рис. 16.3. Выбор параметров в диалоговом окне профилирования

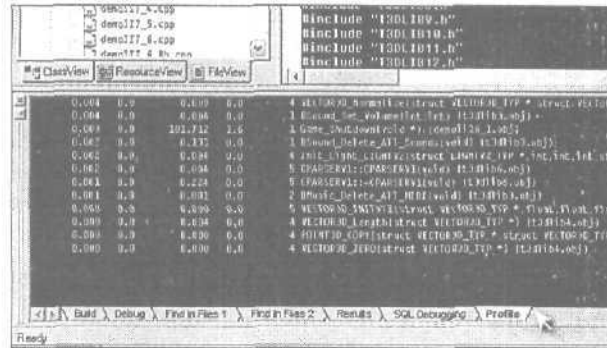


Рис. 16.4. Выбор окна вывода результатов профилирования в Visual C++



Рис. 16.5. Развернуть представление результатов профилирования для проведения анализа данных

## Анализ результатов профилирования

После того как программа-профилировщик была запущена в режиме Function Timing или Function Coverage Mode, на экране появится окно вывода профилирования.

Profile: Function timing, sorted by time  
Date: Fri Jan 03 18:57:23 2003

### Program Statistics

-----  
Command line at 2003 Jan 03 18:55:  
"D:\source\t3dcodeII\test"  
Total time: 12949.659 millisecond  
Time outside of functions: 3.581 millisecond  
Call depth: 6  
Total functions: 574  
Total hits: 12775605  
Function coverage: 20.6%  
Overhead Calculated 5  
Overhead Average 5

### Module Statistics for test.exe

-----  
Time in module: 12946.078 millisecond  
Percent of time in module: 100.0%  
Functions in module: 574  
Hits in module: 12775605  
Module function coverage: 20.6%

Func	Func+Child	Hit	
Time	%	Time	% Count Function

-----  
**2751.80421.3 2751.804 21.3 164604**

**Draw\_Textured\_TriangleGSZB\_16**(struct POLYF4DV2\_TYP \*,  
unsigned char \*,int,unsigned char \*,int) (t3dlib10.obj)

**1115.720 8.6 1115.720 8.6 158**

**DDraw\_Flip2**(void) (t3dlib10.obj)

**1041.832 8.0 1484.683 11.5 316**

**Light\_RENDERLIST4DV2\_World2\_16**(  
struct RENDERLIST4DV2\_TYP\*,struct CAM4DV1\_TYP\*,  
struct LIGHTV2\_TYP \*,int) (t3dlib8.obj)

**653.307 5.0 1334.972 10.3 316**

**Remove\_Backfaces\_RENDERLIST4DV2**(  
struct RENDERLIST4DV2\_TYP \*,struct CAM4DV1\_TYP \*)  
(t3dlib7.obj)

**631.246 4.9 631.246 4.9 962220**

**Insert\_POLY4DV2\_RENDERLIST4DV2**(  
struct RENDERLIST4DV2\_TYP \*,struct POLY4DV2\_TYP \*)  
(t3dlib7.obj)

...

Заголовок содержит общую информацию; однако особый интерес представляют отдельные строки профиля:

- **Func Time** — количество времени, затраченное на выполнение самой функцией;
- **Func Time+ Child Time** — количество времени, затраченное на выполнение функций плюс все вызовы дочерних функций внутри данной родительской;
- **Hit Count** — количество вызовов данной функции.

Хотя данная информация не очень точная, ее вполне можно использовать в качестве отправной точки. Чтобы получить эту информацию самостоятельно, нам пришлось бы проделать значительную работу. В качестве примера рассмотрим первую строку профиля (она выделена). Это вызов растеризатора многоугольников. Как следует из приведенных цифр, на долю этой функции приходится примерно 21% вычислений данного приложения. Следовательно, если ускорить выполнение этой функции в 2 раза, то скорость работы приложения в целом также заметно увеличится, поскольку раздел, **занимавший** 21% времени, будет занимать только 10.5%.

Профилирование кода является **мощным** методом выявления в программе функций, на выполнение которых расходуется больше всего времени, что позволяет сосредоточиться на их оптимизации. Например, хотя вы считаете медленной функцию освещения, может оказаться, что очистка Z-буфера (которая состоит из нескольких строк кода) занимает практически все время, поэтому можно заняться именно этими несколькими строками и получить более существенный выигрыш в скорости, чем в результате переделки 500–1000 строк кода функции освещения. Следует правильно выбирать **направление** приложения сил!

## Оптимизация с помощью VTune

Все слышали об этом средстве, но кто реально использует его? Люди, которые хотят добиться абсолютного результата! VTune компании Intel, вне сомнения, является **одним** из самых мощных средств, которые я когда-либо использовал. К сожалению, его возможности обеспечиваются за счет сложности. Однако компания Intel приложила большие усилия, чтобы максимально упростить использование данного средства начинающими, поэтому некоторыми его возможностями можно пользоваться уже через несколько минут после установки и запуска.

Для тех, кто не слышал о VTune, поясняю, что это, по сути, средство оптимизации программного обеспечения от компании Intel. Оно позволяет просматривать, анализировать и оптимизировать любую программу. С его помощью можно анализировать все: время **выполнения** функций, использование памяти, доступ к кэшу и т.д. На данную тему можно написать целую **книгу**, поэтому пока что закончим обсуждение и попытаемся установить VTune и опробовать его.

### Установка VTune

Сначала необходимо загрузить последнюю версию VTune с Web-узла компании Intel <http://developer.intel.com/software/products/VTune/>

Посмотрите на Web-страницу, найдите ссылку Free Evaluation Download и загрузите программное обеспечение. Убедитесь, что вы выбрали правильную версию — Visual C++ или .NET, в зависимости от используемого вами компилятора. После загрузки приложения запустите инсталлятор и установите VTune на ваш компьютер. Установка производится очень просто, но после установки приложения может понадобиться выполнение перезагрузки.

## Подготовка к профилированию

После успешной установки приложения можно приступить к профилированию. Однако перед запуском VTune необходимо подготовить тестируемое приложение. Мы вновь будем использовать `DEMO116_1.CPP|EXE`, поэтому запускаем Visual C++, загружаем проект с этим приложением, готовым к компиляции. Сначала необходимо установить некоторые настройки компилятора. Для этого выполняются следующие действия.

1. Поскольку мы готовимся профилировать код времени исполнения, лучше выбрать режим окончательной версии программы. Для этого нужно выбрать в меню **Build** → **Set Active Configuration**, а затем выбрать тип вашего приложения так, как показано на рис. 16.6.

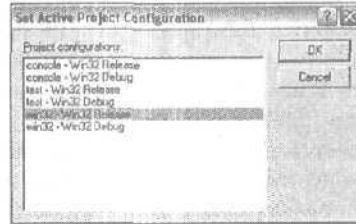


Рис. 16.6. Выбор типа приложения

2. Теперь нужно убедиться, что все символьные имена доступны VTune. Для этого выбираем в главном меню **Project** → **Settings** и во вкладке **C++** выбираем **General** в поле **Category**. Устанавливаем флажок **Generate Browse Info** и в поле **Debug Info** выбираем **Program Database**. Это гарантирует, что профилировщик сможет видеть все символьные имена и исходный код. На рис. 16.7 показано упомянутое диалоговое окно и процесс выбора. Не закрывайте его, мы еще не закончили работу с ним.

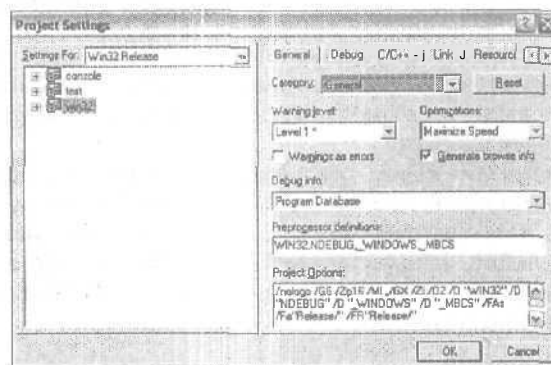


Рис. 16.7. Символьные имена и исходный код должны быть доступны для VTune

3. Нам еще нужно установить некоторые элементы во вкладке **Link**; выбираем эту вкладку, а затем выбираем категорию **General**. Теперь убеждаемся, что флажок **General Debug Info** отмечен. Если вы используете версии **Professional** или **Enterprise** Visual C++, можно также установить флажок **Enable Profiling**, чтобы иметь впоследствии возможность сравнить результаты профилирования разными средствами. Процесс выбора показан на рис. 16.8.



1. После **щелчка** на кнопке Quick Performance Analysis Wizard на экране появится еще одно диалоговое окно, показанное на рис. 16.10. Оно позволяет выбрать профилируемую программу, а также задать условия ее выполнения. В данном случае мы хотим запустить DEMO116\_1.CPP!EXE, для этого нужно щелкнуть на кнопке "...", что приведет к появлению диалога поиска файла и позволит выбрать нашу демонстрационную программу. Затем следует выбрать приложение .EXE, как показано на рис. 16.10.

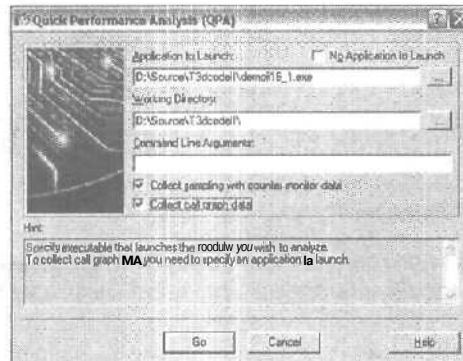


Рис. 16.10. Выбор профилируемого приложения

2. Теперь в исходном диалоге появится имя вашего приложения, его путь и т.д., однако необходимо еще указать тип профилирования. Мы отметим оба флажка: Collect Sampling with Counter Monitor Data и Collect Call Graph Data. Первый задает определение времени, а второй — генерацию полной диаграммы вызовов всех функций.
3. После **щелчка** на клавише Go начнется процесс профилирования. Приложение будет многократно запускаться и останавливаться. Причем последнее выполнение будет очень медленным. В это время происходит окончательный временной анализ. Он может длиться сколь угодно долго; я рекомендую подождать 1-2 минуты и затем просто нажать кнопку Esc или уничтожить демонстрационное приложение, закрыв соответствующее окно. Программа VTune начнет работать, и на экране наконец появится небольшое диалоговое окно, которое позволяет выбрать производимые действия — нужно просто нажать Cancel (рис. 16.11).

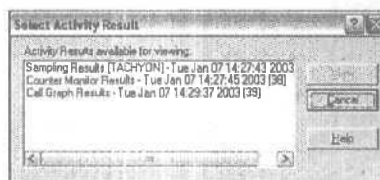


Рис. 16.11. Диалоговое окно выбора результата

Теперь у нас есть полный профиль. Рассмотрим, какую важную информацию можно из него извлечь.

## Анализ данных VTune

Вам необходимо изучить руководство по VTune, чтобы узнать обо всех методах анализа, которые поддерживает данная программа. Рассмотрим некоторые основные моменты, с которых можно начать.

[illegible]

## Анализ представления Clockticks

На данном рисунке также показано, каким будет результат: диаграмма всех процессов, которые выполнялись в ходе анализа, и время их выполнения. Если присмотреться, можно увидеть наше приложение DEMON16\_1.EXE (оно находится внизу и подсвечено). Если дважды щелкнуть на тексте, можно получить более подробное представление нашего приложения, как показано на рис. 16.14.

Это замечательное представление: с его помощью можно узнать, сколько времени уходит на выполнение каждой **функции**. Попробуйте щелкнуть один раз на какой-либо функции диаграммы. В результате она **подсветится**, и вы увидите, как изменится **показатель Selection Summary** в правом **разделе** представления. Это изменение отражает суммарное количество системных тактов, затраченных данной **функцией**, и процентное отношение времени процессора, затраченного на данную функцию, ко времени, затраченному на остальные функции. Если щелкнуть на функции дважды, откроется новое окно, в котором представлен исходный код функции. Чтобы вернуться в режим просмотра функций, достаточно закрыть данное окно или нажать комбинацию клавиш **Ctrl+Tab**.

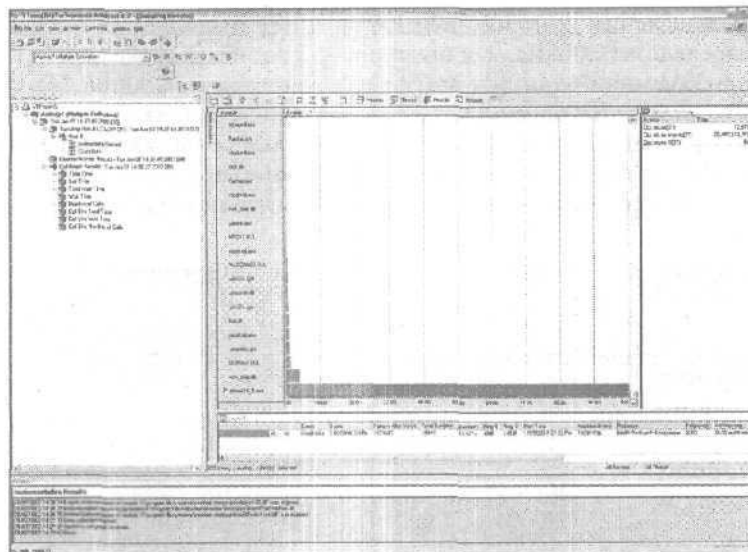


Рис. 16.13. Анализ использования времени процессора на уровне процессов

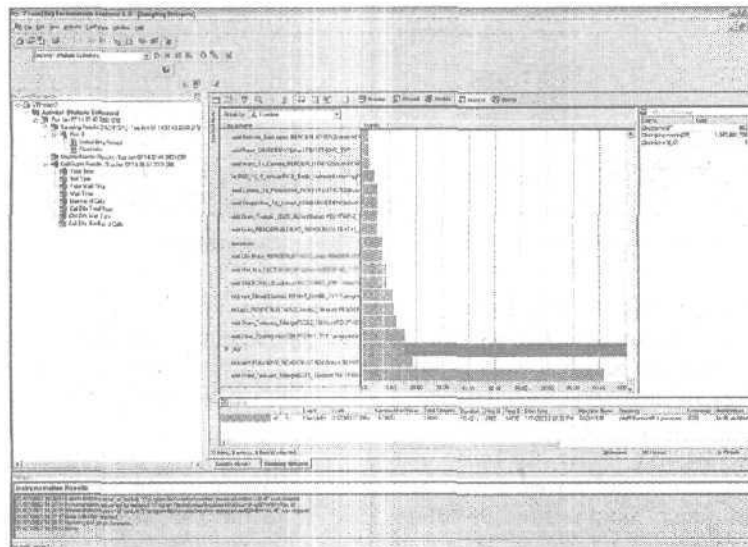


Рис. 16.14. Анализ нашего приложения на уровне функций

Теперь я покажу вам совершенно удивительную вещь. Посмотрите на третью снизу функцию диаграммы (я подсветил ее на рис. 16.14). Это функция `_ftol()`, которая преобразует числа с плавающей точкой в целые. Как следует из диаграммы, единственная C-функция требует практически столько же процессорного времени, что и весь код трехмерного игрового процессора! Это одна из самых непонятных математических проблем — когда некоему целому числу (`int`) присваивается значение с плавающей точкой (`float`), например,

```
float f;
int i=(int)f;
```

компилятор генерирует вызов функции, преобразующей значение с плавающей точкой в целое. Таким образом, если мы даже полностью оптимизируем остальной код, одна лишь эта деталь может свести на нет все наши усилия. Я затрагивал эту тему во многих книгах и позднее покажу некие приемы оптимизации. Однако главным является то, что с помощью данного представления мы видим: если удастся обойтись без этого вызова функции, это позволит сэкономить около 5% компьютерного времени.

В качестве проверки сравним результаты, полученные с помощью встроенного профилировщика Visual C++, с результатами VTune. Выберем в диаграмме функцию `Draw_Textured_TriangleGSZB_16()` и рассмотрим ее сводку в правом разделе экрана, как показано на рис. 16.15.

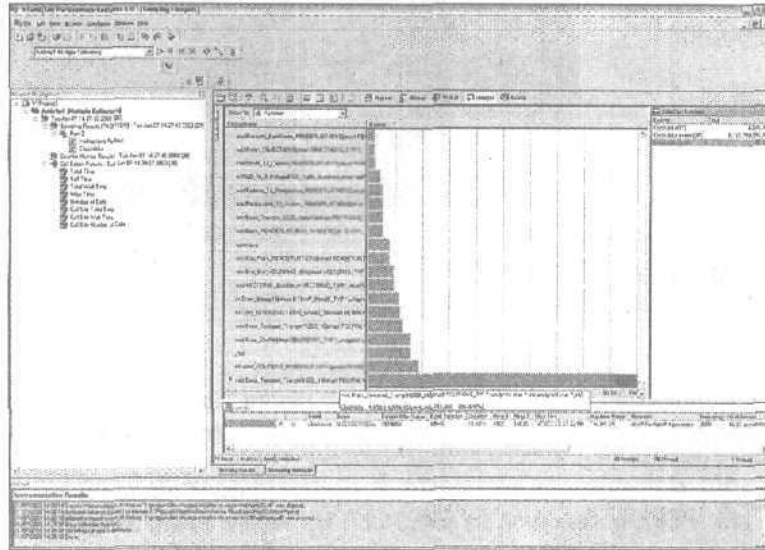


Рис. 16.15. Исследование производительности отдельной функции

Обратите внимание на последнюю строку, в которой указывается процентное отношение времени, которое составляет 35.9%. Согласно информации, предоставленной профилировщиком Visual C++, на эту функцию тратилось 21.3% времени. Данные несколько расходятся. Это зависит от многих факторов, в частности от того, как производились тесты и т.д., так что не следует особо беспокоиться по этому поводу. С помощью диаграммы можно исследовать каждую функцию и (в идеале) определить, на чем следует сосредоточить усилия при оптимизации. Из этого представления можно, естественно, извлечь и другую информацию, однако пока остановимся на достигнутом. Если же вы хотите двигаться далее, вам следует обратиться к руководству по VTune.

#### Анализ диаграммы вызовов

Рассмотрим еще одно важное представление, основанное на диаграмме вызовов программы, и проанализируем, что чем вызывается. Чтобы выбрать представление Call Graph, нужно дважды щелкнуть на строке `Call Graph Results` в самом левом разделе. В результате на экране появится изображение, похожее на представленное на рис. 16.16. Это дерево, которое дает возможность увидеть функции, вызывающие данную, и функции, вызываемые данной. Чтобы открыть или закрыть список вызовов, достаточно просто щелкнуть на маленькой пиктограмме `<and>` в конце каждой функции. Чем более

красным цветом выделена функция, тем больше времени тратится на нее. Наше приложение на рис. 16.16 называется `thread_13A4`.

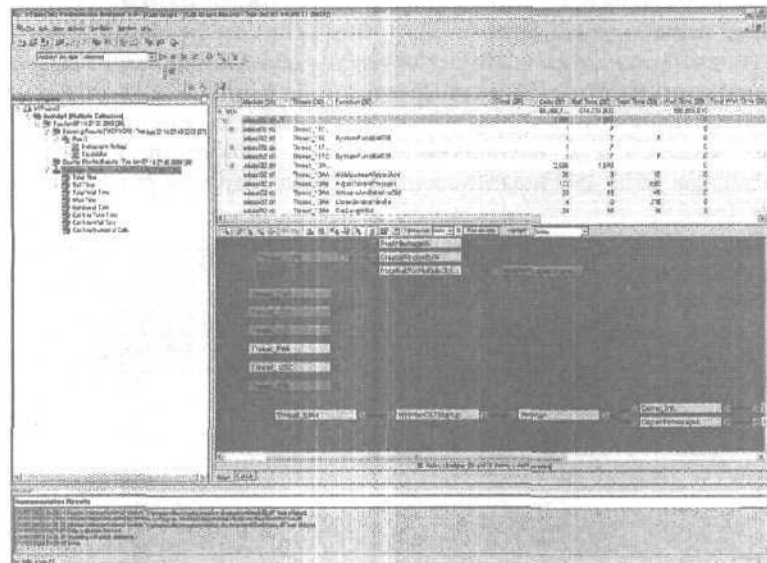


Рис. 16.16. Анализ диаграммы вызовов

На рис. 16.17 показано внутреннее строение функции `Light_RENDERLIST4D()`. Заметим, что эта функция вновь содержит очень много вызовов пресловутой функции `_ftol()`! Фактически именно на нее тратится большая часть времени. Переключимся в другое состояние и посмотрим, какие функции вызывают `_ftol()`. Для этого следует щелкнуть на левой пиктограмме функции `_ftol()`, показанной на рис. 16.17. На экране появится диаграмма вызовов функции `_ftol()` (если ее еще нет на экране), что позволит нам увидеть все функции, из которых желательно удалить этот вызов, применив некий другой метод кодирования.

В заключение хочется еще раз подчеркнуть, что VTune — исключительно мощное средство. Я советую вам поработать с ним несколько часов — пощелкать все, что можно, и посмотреть, что получится, а также обязательно просмотреть обучающую программу, предлагаемую в начале данного приложения.

## Использование компилятора Intel C++

Как известно, 95% всех игр для персональных компьютеров написаны на Visual C++. В этом нет ничего удивительного, поскольку компания Microsoft предлагает действительно стоящие программные средства и удачно встраивает их во многие системы. Visual C++ вполне заслуженно считается очень удачным продуктом, а .NET еще лучшим.

Тем не менее, было время, когда другие компании создавали для персональных компьютеров компиляторы, которые довольно широко использовались. Помните Borland и Watcom? Они вышли из бизнеса или были поглощены, поскольку не смогли конкурировать с Microsoft. Однако компания Microsoft сейчас стала неповоротливым монстром, которому сложно удерживать первенство во всем (хотя она и пытается это делать), и компиляторы как раз являются той областью, в которой компании не удалось создать лидирующий по всем показателям продукт.

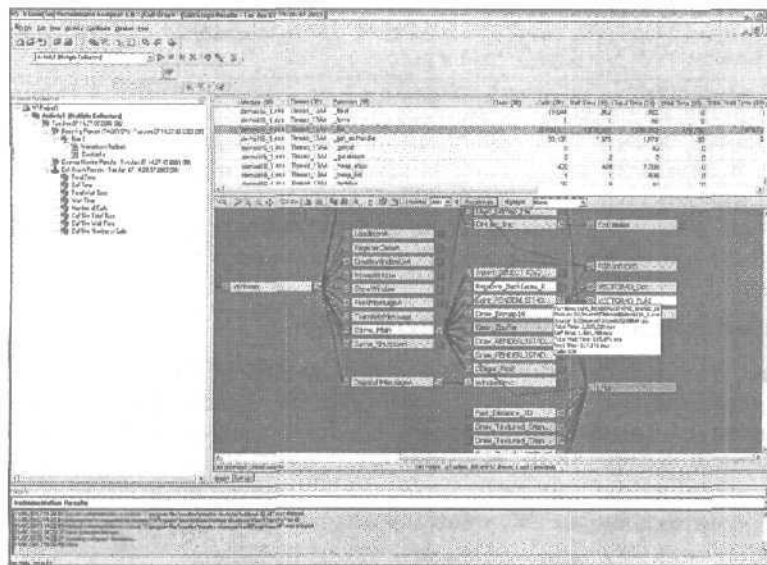


Рис. 16.17. Исследование отдельной функции

Тому есть определенная причина: компания Microsoft разрабатывает программное обеспечение, а не микропроцессоры, поэтому она не может знать о процессорах Pentium абсолютно все, чтобы написать самый совершенный компилятор. Компилятор компании Microsoft является лучшим средством для создания широкого диапазона программного обеспечения от приложений до .DLL и компонентов ActiveX, но он не является наилучшим оптимизатором для процессоров Pentium компании Intel. Лучшим оптимизатором является компилятор самой Intel.

Многие даже не знают, что компания Intel разработала свой компилятор. Это действительно коммерческий компилятор, причем самый лучший оптимизирующий компилятор для процессоров Intel. Это означает, что вы непременно захотите компилировать с его помощью свои игры, если получите такую возможность. Иногда программа станет быстрее, иногда — нет; в любом случае, стоит его попробовать. Если вы создаете коммерческий продукт, лучше потратить на увеличение скорости несколько сотен долларов, чем десятки или сотни человеко-часов, которые необходимы, чтобы достичь такого же увеличения производительности с помощью оптимизации вручную и использования Visual C++.

Сложно заставить себя изучать новое средство после многих лет успешного использования Visual C++, однако компания Intel учла это и сделала свой компилятор C++ встраиваемым в Visual C++ Studio/.NET. Нужно только установить единственный флажок, и все заработает. Может оказаться, что простая перекомпиляция программы с помощью компилятора Intel приведет к увеличению скорости от 0% до 200%, поэтому определенно стоит потратить немного времени на его изучение.

## Загрузка оптимизирующего компилятора Intel

Сначала необходимо загрузить сам компилятор. Последнюю версию можно найти на Web-узле компании Intel по адресу

<http://developer.intel.com/software/products/compilers/>

Найдите на странице ссылку на бесплатный компилятор Intel для Windows и загрузите его на свой жесткий диск.

После загрузки компилятора убедитесь, что все приложения (такие как Visual C++ и VTune) закрыты, и запускайте программу установки. Процесс установки происходит как обычно. В определенный момент на экране появится диалоговое окно, в котором предлагается выбрать поддержку процессора Itanium — *не делайте* этого. Используйте установку **Typical**. Если все в порядке, то установка проходит очень **гладко**, как будто вообще ничего не происходит. Теперь перейдем к самой интересной части.

## Использование компилятора Intel

Запускаем Visual C++ и загружаем в рабочую область DEMOII16\_1.CPP\EXE. Visual C++ загрузится, и на экране, как обычно, появится рабочая область. Теперь внимание: в главном меню выбираем **Tools⇒Select Compiler**. В результате на экран выводится диалоговое окно, показанное на рис. 16.18. Находим маленькое окошко флажка под названием Intel® C++ Compiler и отмечаем его — теперь мы используем оптимизирующий компилятор Intel. Щелкните на кнопке **OK**, чтобы закрыть данное диалоговое окно.

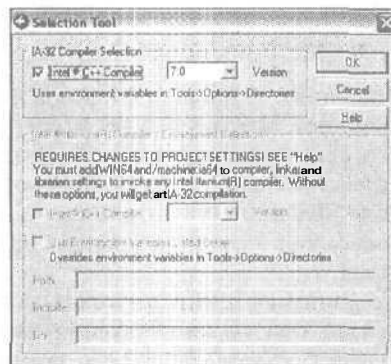


Рис. 16.18. Выбрать компилятор Intel очень легко

99% всех настроек компилятора работают с компилятором Intel аналогично, поэтому ничего менять не нужно. Однако компилятор Intel содержит множество новых **оптимизационных** возможностей, которые не поддерживаются в Visual C++ и которые вам необходимо будет добавить вручную, **установив** соответствующие флаги.

Я покажу, как это делается, однако сначала хочу предложить попытаться скомпилировать и запустить наше демонстрационное приложение DEMOII16\_1.CPP и посмотреть, будет ли оно работать. Выбираем в главном меню **Build⇒Rebuild All**, чтобы выполнить компиляцию проекта. В процессе построения появляются непонятные предупреждения или сообщения компилятора, которых раньше не было, — это хороший знак, так как это означает, что компилятор Intel работает. Когда приложение скомпилировано, мы продолжаем свою работу.

**ВНИМАНИЕ**

Процесс компиляции может длиться в 2-3 раза дольше, чем обычно, поскольку компилятор Intel пытается применить разнообразные оптимизационные стратегии.

Если процесс компиляции пройдет успешно, вы получите несколько предупреждений (это следствие моего немного небрежного стиля программирования), но ошибок быть не должно. Теперь запустим программу и посмотрим, работает ли она. В главном меню вы-

бираем Build⇒ Execute... или нажимаем <Ctrl+F5>, и на экране появятся движущиеся роботы. Если присмотреться внимательней, можно заметить, что анимация выглядит чуть живее. Так и есть. Демонстрационная программа в результате простой перекомпиляции без использования всяких дополнительных возможностей стала выполняться на 5-10% быстрее (в зависимости от технических характеристик вашего компьютера).

## Использование дополнительных возможностей компилятора

Оптимизирующий компилятор Intel имеет много дополнительных оптимизационных возможностей, которыми можно управлять с помощью соответствующих флажков компилятора, устанавливая их вручную. Список некоторых наиболее интересных возможностей представлен в табл. 16.1

**Таблица 16.1. Некоторые опции оптимизирующего компилятора Intel**

Опция	значение
<b>Оптимизация для процессоров</b>	
-G5	Для процессора Pentium
-G6	Для процессоров Pentium Pro, Pentium II, Pentium III
-G7	Для процессоров Pentium 4 и Хеоп
<b>Оптимизация с учетом специфики процессора</b>	
-Qxi	Процессоры Intel Pentium Pro, Pentium II (использующие команды CMOV, FCMOV и FCOMI)
-QxM	Процессоры Pentium с командами технологии MMX (не включая i-команды)
-QxK	Процессор Pentium III с потоковыми расширениями SIMD SSE (включая i- и M-команды)
QxW	Процессоры Pentium 4 и Хеоп с потоковыми расширениями SIMD SSE2 (включая i-, M- и K-команды)
<b>Опция обеспечивает оптимизацию для следующих устройств</b>	
-Qaxi	Для процессоров Intel Pentium Pro, Pentium II (использующих команды CMOV, FCMOV и FCOMI)
-QaxM	Для процессоров Pentium с командами технологии MMX
-QaxK	Для процессора Pentium III с потоковыми расширениями SIMD SSE (включая i- и M-команды)
-QaxW	Процессоры Pentium 4 и Хеоп с потоковыми расширениями SIMD SSE 2 (включая i-, M- и K-команды)
-Qax(i;M;K;W)	обеспечивает возможность использования векторизации и генерирует специализированный обобщенный код IA-32 (более медленный)
-Qx(i;M;K;W)	включает векторизацию и генерирует специализированный, специфичный для данного процессора код (более быстрый)
-O1	Оптимизация по скорости, однако оптимизации, дающие малый выигрыш в скорости, но приводящие к увеличению размеров кода, запрещены

Опция	значение
<b>Общие уровни оптимизации</b>	
-O2	Оптимизация по скорости
-O3	Разрешена высокоуровневая оптимизация (этот уровень не гарантирует более высокую производительность)
-Og	Разрешена глобальная оптимизация
-Os	Разрешено большинство оптимизаций по скорости, однако <b>запрещены</b> оптимизации, дающие малый выигрыш в скорости, но приводящие к увеличению размеров кода
-Ox	Разрешены все оптимизации по скорости

Существует гораздо больше опций, чем перечислено в списке, но пока ограничимся этими. Попробуйте настроить компилятор на характеристики вашего процессора, опробуйте всевозможные способы оптимизации, в том числе с возможностью векторизации. Однако учтите, что разрешение векторизации может привести к повреждению кода. Если это происходит, всегда можно скомпилировать отдельные файлы исходного кода с различными опциями или просто использовать для таких файлов компилятор Microsoft. Покажем, как заставить Visual C++ использовать для одних файлов исходного кода компилятор Microsoft, а для других — компилятор Intel.

## Выбор компилятора для исходных файлов

Предположим, что в диалоговом окне Select Compiler был выбран компилятор Intel, однако некий исходный файл дает слишком много ошибок при компиляции данным компилятором. Если вы хотите, чтобы для этого конкретного файла (или любого другого) использовался стандартный компилятор Microsoft, можно применить следующую процедуру.

1. В окне, содержащем перечень файлов исходного кода, щелкнуть правой кнопкой мыши на **соответствующем** файле и выбрать Settings.
2. На экране появится диалоговое окно, представленное на рис. 16.19. Выберите вкладку C/C++, а в поле Category выберите **General**.

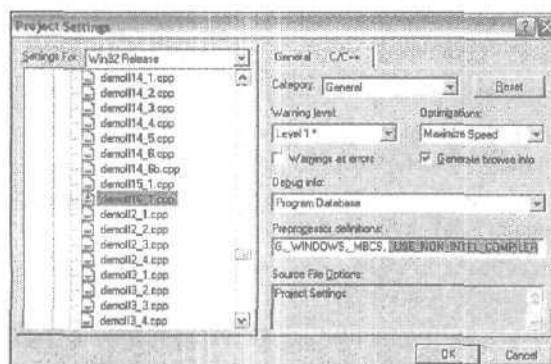


Рис. 16.19. Выбор компиляторов Intel/Visual C++ для отдельных файлов

3. В поле ввода Preprocessor Definitions ввести следующую константу.

`_USE_NON_INTEL_COMPILER`

Это приведет к тому, что Visual C++ будет использовать компилятор Microsoft вместо компилятора Intel.

СОВЕТ

Можно также в качестве основного указать компилятор Microsoft, а затем с помощью макроопределения `_USE_INTEL_COMPILER` переключиться на использование компилятора Intel для определенных исходных файлов.

## Стратегии оптимизации

После того как вы заставите компилятор работать и сможете с его помощью компилировать свои программы, опробуйте все возможные варианты, разрешая и запрещая различные способы оптимизации, отмечая определенные процессоры, а затем используйте профилирование или программу **VTune**, чтобы найти узкие места. Естественно, это следует делать логичным и последовательным образом — исследовать опции компилятора по одной и фиксировать полученные результаты. Если в каждом эксперименте производить сразу несколько изменений, невозможно будет отследить, что к чему приводит, поэтому следует за один раз менять только одну опцию и фиксировать полученные результаты.

## Введение в SIMD

Тема, к рассмотрению которой я приступаю, вызывает у меня особые чувства. Это **SIMD-программирование** (Single Instruction Multiple Data — архитектура с одним потоком команд и несколькими потоками данных). Если вы знакомы с SIMD и используете его, можете пропустить данный раздел; если же нет, полученная информация будет, безусловно, полезна для вас.

Сначала поговорим о терминологии, хронологии и исторических фактах. После выпуска первой партии процессоров Pentium компания Intel выпустила процессор с расширениями мультимедиа (Multimedia Extensions, MMX). Ожидалось, что новая технология MMX изменит мир, позволит ускорить трехмерную графику и доступ в Internet — словом, была обычная рекламная шумиха.

Однако MMX не стала популярной и не оправдала ожиданий. Причина в том, что MMX неудачно встроена в архитектуру процессора Pentium. По существу, технология MMX добавила несколько дополнительных инструкций в набор инструкций Pentium, чтобы позволить выполнять параллельные или SIMD операции с целыми числами.

У Intel просто не было регистров, которые можно было выделить под рабочие регистры MMX, поэтому было принято решение расположить их в регистрах с плавающей точкой (FPU) и использовать 64 бита из 80, выделенных каждому регистру FPU, как показано на рис. 16.20а.

Возможно, это была самая большая глупость в истории микропроцессорной архитектуры. Ведь для того, чтобы использовать команды MMX, нужно было переключить состояние процессора, сохранить регистры FPU, выполнить необходимые математические операции, а затем вернуть все в прежнее состояние. Поэтому, несмотря на возможность параллельно выполнять сложения с помощью MMX, из-за переключения режимов и восстановления состояний в целом все получалось только медленней! Таким образом, от MMX не было никакой пользы.

Однако при переходе к Pentium III компания Intel учла совершенные ошибки и сделала все как следует, реализовав Streaming SIMD Extensions (поточковые расширения SIMD, SSE).

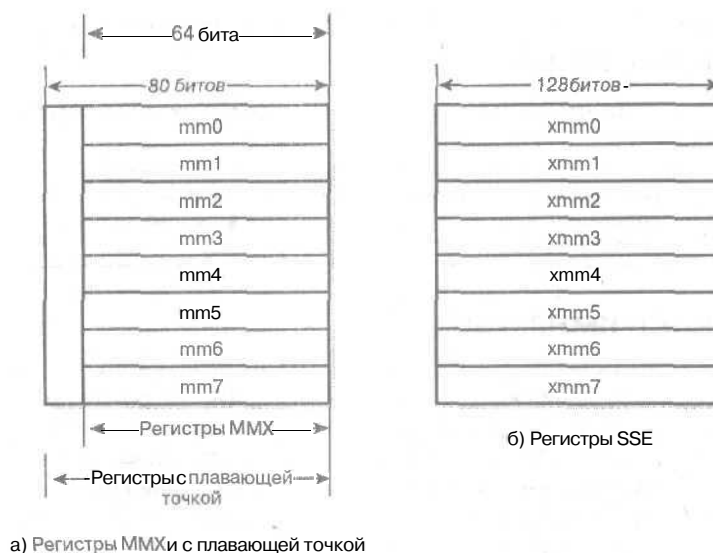


Рис. 16.20. Размещение регистров MMX в стеке FPU

Для реализации SSE компания Intel предусмотрела восемь совершенно новых 128-битовых регистров, пронумерованных от `xmm0` до `xmm7`, как показано на рис. 16.20б. Кроме того, SSE может функционировать параллельно с FPU и даже с целочисленными регистрами MMX, и при этом не требуется выполнять переключения контекста, сохранять информацию и т.п. В каждом из новых регистров может храниться до четырех 32-битовых чисел с плавающей точкой с одинарной точностью. Таким образом, SSE — это реализация SIMD для чисел с плавающей точкой, которая дает возможность одновременно обрабатывать четыре 32-битовых значения с плавающей точкой. Можно выполнять сложение, умножение, извлекать квадратные корни, производить сравнения и т.д. Фактически к набору команд процессора было добавлено около 70 новых инструкций для SSE — это все равно, что иметь еще один процессор внутри процессора.

Прежде чем двигаться дальше, следует упомянуть вторые потоковые расширения SIMD (Streaming SIMD Extensions 2, SSE2). Это новые дополнения к процессору Pentium 4, содержащие 150 новых инструкций, которые обеспечивают поддержку операций с плавающей точкой для чисел с двойной точностью. Таким образом, процессор Pentium 4 может выполнять параллельно две операции с 64-битовыми значениями с двойной точностью вместо четырех операций с 32-битовыми значениями с одинарной точностью, как показано на рис. 16.21.

SSE2 обладает большими возможностями, но если вы не занимаетесь вычислениями, требующими 64-разрядной точности, то значения с двойной точностью вам не нужны, и лично я предпочел бы одновременное выполнение четырех операций с помощью SSE вместо двух операций с двойной точностью в SSE2. Однако Pentium 4 поддерживает как SSE, так и SSE2, поэтому код, написанный для Pentium III, будет отлично работать на Pentium 4. Кроме того, SIMD-регистры `xmm0`–`xmm7` в обеих реализациях (SSE и SSE2) содержат 128 разрядов и работают одинаково. Поэтому мы обсудим только SSE в Pentium III; иными словами, только 32-разрядные четырехканальные SIMD-вычисления с одинарной точностью.

Я думаю, теперь вам понятно, почему SSE/SSE2 так редко используется — ведь сначала нужно разобраться, что означают все эти сокращения :-). Начиная с этого момента я

буду называть SSE/SSE2 просто SIMD, а для того, чтобы все предлагаемые программы работали, нужен процессор Pentium III или более поздний.



Рис. 16.21. Разбиение 128-разрядного регистра в SSE/SSE2

## Базовая архитектура SIMD

Базовая архитектура SIMD состоит из восьми 128-битовых регистров, как показано на рис. 16.20б. Каждый регистр содержит четыре 32-битовых значения с одинарной точностью и операции с этими значениями проводятся параллельно. Каждый из регистров XMM0–XMM7 имеет формат, показанный на рис. 16.21: биты 0–31 представляют младший элемент, а биты 96–127 — старший. Напомню, что вычисления с этими элементами производятся параллельно, отсюда и название: одна команда, много данных (single instruction, multiple data, SIMD).

Архитектура SIMD идеально подходит для трехмерной графики, поскольку многие операции в ней можно распараллелить (вычисление скалярных произведений, умножение матриц, освещение, растеризацию и т.п.). Однако не стоит возлагать слишком большие надежды: хотя SIMD позволяет выполнять за один раз четыре операции с плавающей точкой, отправка данных в регистры и извлечение из них, а также приведение данных программы к подходящей для SIMD форме приводят к снижению производительности. В лучшем случае выигрыш в производительности составит в среднем 200%, при достаточно рациональном программировании SIMD он составляет 150–170%. Таким образом, математическая часть вашего кода в большинстве случаев будет выполняться, как минимум, вдвое быстрее.

## Работа с SIMD в реальном мире

Использовать SIMD очень просто, если знать этапы, которые мы сейчас опишем. Во-первых, в дополнение к процессору Pentium III необходима поддерживающая SIMD операционная система. Windows 2000, XP и более поздние версии поддерживают SSE/SSE2. Однако даже если в вашем распоряжении находится Pentium III, поддерживающий SIMD, всегда необходимо проверить операционную систему, чтобы определить, поддерживает ли она SSE. В среде Windows проверку поддержки MMX и SSE можно выполнить с помощью следующего кода.

```
if (IsProcessorFeaturePresent(PF_MMX_INSTRUCTIONS_AVAILABLE))
    printf("\nMMX Available.\n");
```

```

else
    printf("\nMMXNOT Available.\n");

if (IsProcessorFeaturePresent(PF_XMMI_INSTRUCTIONS_AVAILABLE))
    printf("\nXMMI Available.\n");
else
    printf("\nXMMI NOT Available.\n");

```

Здесь используется вызов функции Windows API `IsProcessorFeaturePresent()`, которая возвращает TRUE или FALSE на вопрос о конкретном запрашиваемом свойстве. Данная функция позволяет узнать о наличии многих других свойств; советую обратиться к подсказке вашего API, чтобы увидеть их список.

После того как вы выясните, что операционная система и процессор поддерживают SSE (`PF_XMMI_INSTRUCTIONS_AVAILABLE`), необходимо написать программу, использующую SSE, или воспользоваться готовой. Компилятор Intel можно настраивать на использование возможностей Pentium III и 4, т.е. он готов использовать SIMD в процессе оптимизации. Однако мы заинтересованы в том, чтобы написать свой собственный код. Посмотрим, что нам для этого нужно.

## Настройка компилятора для поддержки SIMD

Оптимизирующий компилятор Intel осуществляет поддержку SIMD и содержит соответствующие заголовочные файлы и библиотеки. Однако если вы используете Microsoft Visual C++, необходимо загрузить пакет обновлений для последних версий процессоров. В результате будут установлены все необходимые заголовочные файлы, библиотеки и т.д. Вам понадобится Visual C++ Professional или Enterprise, а также четвертая или пятая версия Service Pack (я советую использовать пятую или более позднюю версию). Пакет обновлений для последних версий процессоров вы найдете по адресу

<http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.asp>

СОВЕТ

В результате установки данного пакета вы получите свежую версию MASM.

Я настоятельно рекомендую сначала установить последнюю версию Service Pack для Visual C++ Studio, находящуюся по адресу

<http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.asp>

СОВЕТ

Если в вашем распоряжении нет Visual C++ Professional или Enterprise, но вы хотите поработать с SIMD-командами, не беспокойтесь: они встроены в компилятор Intel, поэтому достаточно загрузить бесплатную версию данного компилятора, и все будет работать (по крайней мере, 30 дней).

После того как вы загрузили пакет обновлений для процессора, установите его на вашем компьютере и следуйте командам. Возможно, после завершения установки потребуется перезагрузить компьютер. Теперь вы готовы к работе.

Проверим новый пакет процессора или поддержку SIMD с помощью небольшой программы. Сначала создадим простое консольное приложение, а затем внесем в него следующий код из DEMOII16\_2.CPP.

// DEMOII16\_2.CPP - демонстрационная SIMD -программа

```

#define WIN32_LEAN_AND_MEAN
#include<windows.h> // Включение функциональности Windows
#include<windowsx.h>

```

```

#include <stdio.h>
#include <math.h>
#include <xmmintrin.h> // Необходимо для поддержки
                        // SIMD (SSE I)

// MAIN //////////////////////////////////////////////////////////////////////

void main()
(
    // Вывод данных о поддержке операционной
    // системой/процессором возможностей SIMD
    if (IsProcessorFeaturePresent(
        PF_MMX_INSTRUCTIONS_AVAILABLE))
        printf("\nMMX Available.\n");
    else
        printf("\nMMXNOT Available.\n");

    if (IsProcessorFeaturePresent(
        PF_XMMI_INSTRUCTIONS_AVAILABLE))
        printf("\nXMMI Available.\n");
    else
    {
        printf("\nXMMINOT Available.\n");
        return;
    } // if

    // Определим 3 упакованных значения SIMD (они всегда
    // должны использовать 16-байтовое выравнивание)
    __declspec(align(16)) static float x[4] = {1,2,3,4};
    __declspec(align(16)) static float y[4] = {5,6,7,8};
    __declspec(align(16)) static float z[4] = {0,0,0,0};

    // Небольшой SIMD-код, который суммирует X и Y и
    // сохраняет результаты в Z
    __asm
    {
        movaps xmm0, x // Помещаем значение x в XMM0
        addps xmm0, y // Прибавляем значение y к XMM0
        movaps z, xmm0 // Сохраняем результаты из XMM0 в Z
    } // asm

    // Вывод результатов
    printf("\n x[%f,%f,%f,%f]", x[0], x[1], x[2], x[3]);
    printf("\n+y[%f,%f,%f,%f]", y[0], y[1], y[2], y[3]);
    printf("\n");
    printf("\n= z[%f,%f,%f,%f]\n", z[0], z[1], z[2], z[3]);
} // main

```

Теперь программу следует скомпилировать с **ПОМОЩЬЮ** компилятора Intel или Visual C++ с установленным дополнительным пакетом для последних типов процессоров. Компиляция должна пройти без ошибок. После этого можно выполнить программу; в результате одновременно будут просуммированы два набора из четырех значений с плавающей точкой; вывод программы показан на рис. 16.22.

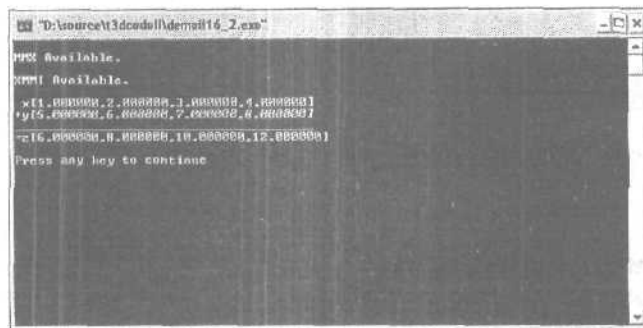


Рис. 16.22. Вывод демонстрационной программы, использующей возможности SIMD

НА ЗАМЕТКУ

Возможно, вы обратили внимание на использование команды `_asm`. Она служит для оповещения компилятора о том, что вы хотите использовать встроенный ассемблерный код. Соответствующие команды просто заключаются в фигурные скобки `{>}`. Более полную информацию о возможностях встроенного ассемблера можно получить из справочной системы вашего компилятора. Вы можете в большей или меньшей степени использовать имена переменных из C/C++, указатели, константы, макросы и т.д. Встроенный ассемблер достаточно “умный” для этого.

Демонстрационная программа сначала создает три массива чисел с плавающей точкой. Каждый массив содержит четыре элемента— эти элементы являются единицами, которые обрабатываются параллельно архитектурой SIMD. Затем программа использует SIMD-команду `movaps`, чтобы переместить 128-битовое значение как единое целое из массива `x[]` в SIMD-регистр `XMM0`, после чего с помощью команды `addps` прибавляет значения `y[]` к `XMM0` и записывает результаты в массив чисел с плавающей точкой `g[]`. Далее мы подробно рассмотрим, что означают эти команды и как они работают; сейчас же важно понять, что с помощью одной команды мы выполнили четыре сложения 32-битовых значений с плавающей точкой. Это просто потрясающе! Попробуйте запустить цикл 1 000 000 раз и измерьте время выполнения, а затем напишите то же самое на C/C++ или даже на FPU-ассемблере и посмотрите, удастся ли вам улучшить результат, полученный с помощью SIMD. Я уверен, что не удастся.

НА ЗАМЕТКУ

Если вы не хотите вводить код вручную, можно использовать код программы `DEMO16_2.CPP` с компакт-диска.

## Типы данных, упаковка и стратегии SOA/AOS

SIMD-команды можно писать на ассемблере, однако предусмотрена и внутренняя библиотека, написанная компанией Intel, которая поддерживается как средой Visual C++, так и компилятором Intel. Первым важным свойством SIMD является новый 128-битовый тип данных `_m128`. Этот тип данных по сути представляет собой упакованный массив из четырех элементов с плавающей точкой, всегда с 16-байтовым выравниванием. При использовании внутренних функций необходимо использовать тип данных `_m128`; в противном случае придется выполнять приведение типов результатов. Однако это практически эквивалентно следующей записи.

```
_declspec(align(16)) float _m128[4];
```

По существу, в SIMD используется так называемый *упакованный* тип данных; т.е. четыре 32-битовых значения с плавающей точкой плотно *упакованы*, начиная с адреса, кратного 16

байтам. Можно использовать или встроенный тип данных `_m128`, или выровненный массив `float[4]`; массив `float[4]` может оказаться более удобным, поскольку позволяет осуществлять доступ к каждому отдельному элементу посредством индексации. Однако чтобы использовать библиотеки встроенных SIMD-команд, необходимо использовать тип `_m128` или выполнить приведение к нему. Далее мы кратко рассмотрим эти встроенные операции.

Следующее важное решение, которое необходимо принять при использовании SIMD, — это способ размещения структур данных. SIMD может обрабатывать четыре элемента данных одновременно и выполнять простые логические или математические операции с ними. Однако при всей эффективности SIMD может оказаться, что процесс преобразования данных в упакованный формат SIMD и извлечение данных после вычислений приводит к напрасной трате времени. Поэтому при определении структур данных следует рассматривать оба формата, массив структур (*array-of-structures, AoS*) и структуру массивов (*structure-of-arrays, SoA*), как показано на рис. 16.23.

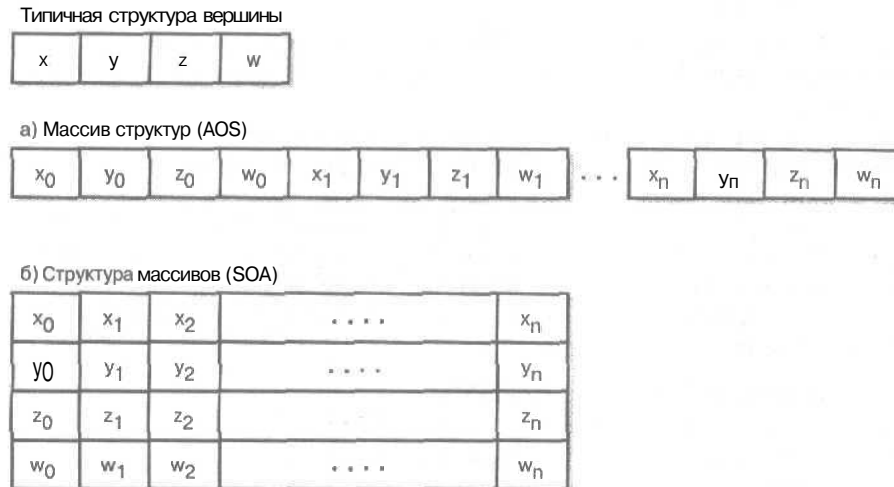


Рис. 16.23. Массив структуры структура массивов

В качестве примера рассмотрим выполнение операций с однородными трехмерными векторами (точками, вершинами). Типичная структура данных может выглядеть следующим образом.

```
typedef struct VECTOR4D_TYP
{
    __declspec(align(16)) float x,y,z,w;
} VECTOR4D, *VECTOR4D_PTR;
```

Используя данную структуру, можно определить 1000 вершин.

```
VECTOR4D vertex_list[1000];
```

Полученная структура данных представляет собой массив структур. Предположим, что мы хотим сложить все вектора. Звучит просто: загрузим некий VECTOR4D в SIMD-регистр, сложим его с другим VECTOR4D, сохраним результат, и так же просуммируем все 1000 векторов.

Забудем на время о деталях и проанализируем, что мы делаем. Мы загружаем в регистр отдельный VECTOR4D в формате  $x_i, y_i, z_i, w_i$  (от младших битов к старшим) и суммируем

его с другим вектором в формате  $x_i y_i z_i w_i$ . Однако при этом теряется целый блок данных, выделенный для  $w$ . Координата  $w$  используется для других целей, а при сложении векторов в большинстве случаев предполагается, что она все время равна 1.0. Это всего лишь один из примеров, демонстрирующий, что иногда лучше использовать другой формат.

Использование структуры массивов позволяет решить данную проблему. Вместо того чтобы определять вершину, а затем составлять из вершин массив, определяем следующую структуру.

```
typedef struct VERTEXLIST4D_TYP
```

```
{
    __declspec(align(16)) float x[1000];
    __declspec(align(16)) float y[1000];
    __declspec(align(16)) float z[1000];
    __declspec(align(16)) float w[1000];
} VERTEXLIST4D, * VERTEXLIST4D_PTR;
```

Затем в цикле вычислений загружаем сразу четыре значения  $x$ , складываем их со следующим набором значений  $x$  и так далее 1000/4 раз. Потом таким же образом складываем значения  $y$  и  $z$ .

Выбор структуры зависит от того, что вы намерены делать с помощью SIMD. Если нужно вычислять много скалярных произведений, то лучше подойдет формат SoA, но для большого количества сложений простых четырехзначных 32-битовых слов лучше использовать AoS. Это весьма важный момент, который необходимо учитывать в SIMD-программировании, а также при разработке структур данных. Например, использование структуры данных SoA позволит ускорить выполнение сложения, но такая структура плохо согласуется со способом хранения данных в нашем игровом процессоре.

## Несколько простых примеров SIMD

**SIMD** — обширная тема. По сути, это целый новый язык ассемблера процессора, в нем поддерживается более 70 инструкций, и все их просто невозможно описать на нескольких страницах. Поэтому мы рассмотрим только выполнение основных математических операций с помощью SIMD.

Встроенная библиотека Intel по сути дублирует большинство SIMD-команд на языке ассемблера. Вероятно, использовать встроенные библиотечные функции проще, чем ассемблерные версии SIMD-команд, но это несколько медленнее, поэтому у вас может возникнуть желание перейти на язык ассемблера. Для того чтобы выполнить любую программу из следующих примеров, необходимо в дополнение к Visual C++ установить пакет обновлений для новых процессоров или использовать компилятор Intel. Кроме того, нужно включить следующий заголовочный файл:

```
#include <xmmintrin.h>
```

**СОВЕТ**

Если вы хотите обеспечить поддержку SIMD-команд с двойной точностью в Pentium 4, необходимо также включить файл `emmintrin.h`.

Наконец, работая с примерами, следует учитывать, что здесь рассмотрены только представители основных классов инструкций.

Во всех примерах используются следующие структуры данных.

```
__declspec(align(16)) float x[4], y[4], z[4];
__m128 m0, m1, m2;
```

## Перемещение и перемешивание данных

Существует множество способов извлечь SIMD-значение в SIMD-регистр с помощью языка ассемблера или встроенных инструкций. В общем случае встроенный ассемблер знает тип каждой переменной и различие между указателем и значением. Таким образом, всегда можно напрямую использовать переменные, которые хранят SIMD-значения, и присваивать их регистрам XMM с помощью команды перемещения выровненных значений с одинарной точностью movaps, имеющей две основные формы:

```
movaps xmm_dest, xmm_src/memory
```

или

```
movaps xmm_dest/memory, xmm_src
```

В принципе, всегда можно переместить содержимое из одного XMM-регистра в другой XMM-регистр. Кроме того, можно перемещать значения из памяти в XMM-регистр или из XMM-регистра в память, но не из памяти в память. Рассмотрим несколько примеров.

Дано:

```
_declspec(align(16)) float x[4], y[4], z[4];
__m128 m0, m1, m2;
```

Пример. Поместить x[] в xmm0

```
_asm
{
    movaps xmm0, x
}
```

Можно также выполнить приведение указателей, чтобы гарантировать, что исходный операнд не вызовет проблем у компилятора.

```
_asm
{
    movaps xmm0, XMMWORD PTR x
}
```

Можно использовать один из указателей данных — esi, edi, edx и т.д.

```
_asm
(
    lea esi, x
    movaps xmm0, [esi]
)
```

Аналогично, чтобы сохранить результат из регистра XMM, просто используется указатель места назначения в памяти.

Пример. Перемещение содержимого xmm0 в z[]:

```
_asm
{
    movaps z, xmm0
}
```

Встроенные библиотечные функции сохранения и извлечения выглядят следующим образом.

```
// Сохраняет значение встроенного типа SIMD __m128
// в массив с плавающей точкой *v
void _mm_store_ps(float *v, __m128 a);
```

```
// Присваивает значения a,b,c,d упакованному встроенному
// типу __m128
__m128 _mm_set_ps(float a, float b, float c, float d);
```

Пример. Инициализировать `m0` значениями `[1,2,3,4]`.

```
m0 = _mm_set_ps(1,2,3,4);
```

Пример. Сохранить `m0` в `x[]`.

```
_mm_store_ps(x, m0);
```

Существует множество других встроенных функций перемещения, инициализации и присваивания, но для начала достаточно уже приведенных.

Теперь, когда мы научились перемещать данные и присваивать **SIMD-данные** регистрам, остановимся на перераспределении или перемешивании данных **SIMD-регистров**. Речь идет о перемещении находящихся в **SIMD-регистрах** слов в другие ячейки внутри некоего **SIMD-регистра**. В наборе **SIMD-команд** существует несколько команд, позволяющих выполнять подобные операции. Мы рассмотрим только одну из них, **shufps**, графически представленную на рис. 16.24. Команда имеет следующий формат.

```
shufps xmm_dest, xmm_src, control8
```

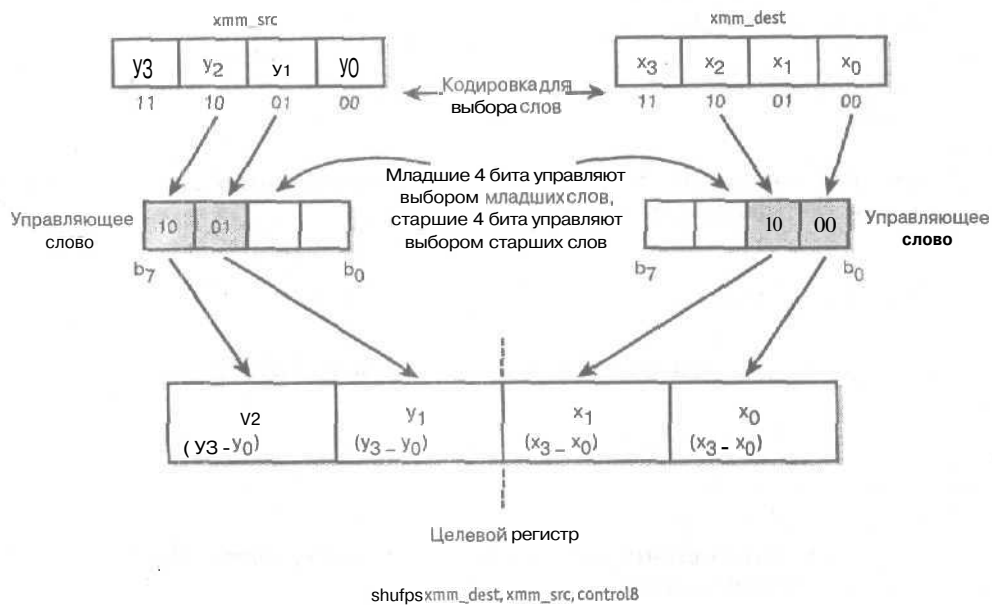


Рис. 16.24. Команда перемещения данных **shufps**

Команда имеет три операнда: источник, адресат и 8-битовое управляющее слово. На значение команды — взять два из четырех слов целевого регистра и два из четырех слов источника и записать их в целевой регистр. Таким образом, целевой регистр используется одновременно в качестве источника. Удобней представить себе выполнение данной команды в виде двух этапов: сначала оба операнда выступают источниками, а затем конечный результат записывается в один из них. В данном случае в прототипе команды это первый операнд, `xmm_dest`. Обсудим данную команду более подробно.

`xmm_dest` — это целевой **SIMD-регистр**, а `xmm_src` — исходный **SIMD-регистр**. Команда производит следующие действия: берутся два слова из четырех, содержащихся

в исходном регистре, и помещаются на место двух старших слов целевого регистра, а также два слова из четырех, содержащихся в целевом регистре, и помещаются на место двух младших слов целевого регистра (рис. 16.24). Таким образом, как уже отмечалось, целевой регистр используется и в качестве источника, и в качестве адресата. Можно считать, что сначала оба регистра (и источник, и целевой регистр) выступают в качестве источников, а окончательный результат в конце процесса записывается в один из этих регистров. Кодирование битов слова, управляющего выбором конкретного 32-битового слова из источника или адресата, представлено в табл. 16.2

**Таблица 16.2. Кодирование управляющего слова операции shufps**

Двухбитовое Значение	Действие
00	Выбирается слово 0 (младшие биты, 0–31)
01	Выбирается слово 1 (биты 32–63)
10	Выбирается слово 2 (биты 64–95)
11	Выбирается слово 3 (старшие биты 96–127)

Кодировка слова `control8` выглядит следующим образом.  
(старший бит)  $A_1A_0$   $B_1B_0$   $C_1C_0$   $D_1D_0$  (младший бит)

Каждый 2-битовый слог определяет выбор одного слова из операндов-источников. В табл. 16.3 показана кодировка каждого управляющего элемента слова.

**Таблица 16.3. Значение 2-битовых элементов управляющего слова операции shufps**

Биты	Действие
$D_1D_0$	Выбирается 32-битовое слово из операнда-адресата и помещается в слово 0 (младшие биты) результата
$C_1C_0$	Выбирается 32-битовое слово из операнда-адресата и помещается в слово 1 результата
$B_1B_0$	Выбирается 32-битовое слово из операнда-источника и помещается в слово 2 результата
$A_1A_0$	Выбирается 32-битовое слово из операнда-источника и помещается в слово 3 (старшие биты) результата

При использовании команды `shufps` вычисление битов, в которые и из которых перемещается слово, может приводить к возникновению ошибок. Решить эту проблему можно с помощью следующей макрокоманды.

```
// srch - A1A0
// srcl - B1B0

// desth - C1C0
// destl - D1D0
#define SIMD_SHUFFLE(srch,srcl,desth,destl) (((srch) << 6) \
| ((srcl) << 4) | ((desth) << 2) | ((destl)))
```

По существу, данная макрокоманда просто получает четыре параметра (слова управления выбором слов из источника и адресата) и соединяет их вместе. Как следует из комментариев, кодирование данных параметров совпадает с кодированием битов в табл. 16.3.

## Поток команд SIMD

В следующих разделах мы обсудим некоторые основные команды **SIMD**. В общем случае все они выполняются параллельно, как показано на рис. 16.25, т.е. одна команда применяется к двум **SIMD**-регистрам, при этом один из регистров используется в качестве целевого.

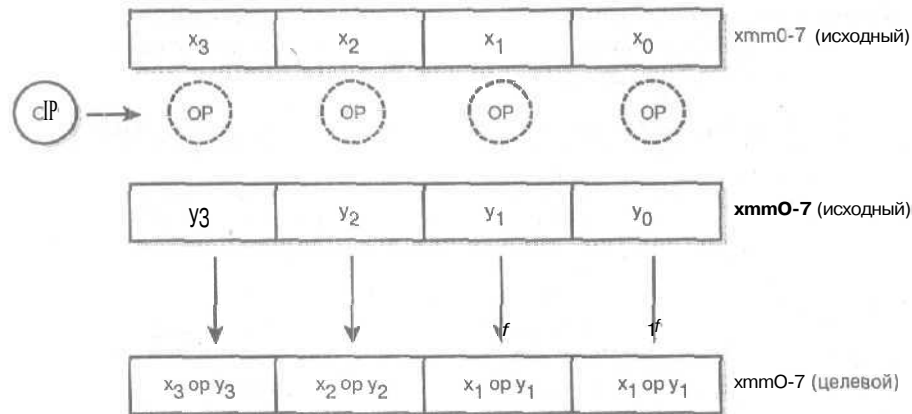


Рис. 16.25. Обобщенные операции SIMD

## Сложение и вычитание

Команды сложения и вычитания упакованных выровненных величин с одинарной точностью позволяют параллельно выполнять сложение и вычитание четырех 32-битовых значений. Начнем со сложения.

`addps xmm_dest, xmm_src/memory`

Операция `addps` прибавляет содержимое некоего регистра **XMM** или 128-разрядного операнда в памяти к содержимому целевого **XMM** регистра.

Пример. Вычисление  $x_0+y_0$ ,  $x_1+y_1$ ,  $x_2+y_2$ ,  $x_3+y_3$ .

```
_asm {
    movaps xmm0, x // Перемещение содержимого x в xmm0
    addps xmm0, y // Прибавление содержимого y к xmm0
    movaps z, xmm0 // Сохранение результатов
    // (x0+y0, x1+y1, x2+y2, x3+y3) в r
}
```

Команда вычитания имеет следующий вид.

`subps xmm_dest, xmm_src/memory`

Операция `subps` вычитает содержимое некоего регистра **XMM** или 128-разрядного операнда в памяти из содержимого целевого **XMM** регистра.

Пример. Вычисление  $x_0-y_0$ ,  $x_1-y_1$ ,  $x_2-y_2$ ,  $x_3-y_3$

```
_asm {
    movaps xmm0, x // Перемещение содержимого x в xmm0
    subps xmm0, y // Вычитание содержимого y из xmm0
    movaps z, xmm0 // Сохранение результатов
    // (x0-y0, x1-y1, x2-y2, x3-y3) в r
}
```

## Умножение и деление

Команды умножения и деления для упакованных выровненных величин с одинарной точностью позволяют параллельно выполнять умножение и деление для четырех 32-битовых значений. Начнем с умножения.

```
mulps xmm_dest, xmm_src/memory
```

Операция `mulps` умножает содержимое некоего регистра XMM или 128-разрядного операнда в памяти и содержимое целевого XMM регистра.

Пример. Вычисление  $x_0 \cdot y_0, x_1 \cdot y_1, x_2 \cdot y_2, x_3 \cdot y_3$

```
_asm {
    movaps xmm0, x // Перемещение содержимого x в xmm0
    mulps xmm0, y // Умножение содержимого y на xmm0
    movaps z, xmm0 // Сохранение результатов
                    // (x0*y0, x1*y1, x2*y2, x3*y3) в r
}
```

Команда деления имеет следующий вид.

```
divps xmm_dest, xmm_src/memory
```

Операция `divps` выполняет деление первого операнда на содержимое некоего регистра XMM или 128-разрядного операнда в памяти и заносит результат в целевой XMM регистр.

Пример. Вычисление  $x_0/y_0, x_1/y_1, x_2/y_2, x_3/y_3$

```
_asm {
    movaps xmm0, x // Перемещение содержимого x в xmm0
    divps xmm0, y // Деление на содержимое y
    movaps z, xmm0 // Сохранение результатов
                    // (x0/y0, x1/y1, x2/y2, x3/y3) в z
}
```

## Извлечение квадратных корней

Команда извлечения квадратного корня для упакованных выровненных величин с одинарной точностью позволяет параллельно извлекать квадратные корни из четырех 32-битовых значений.

```
sqrtps xmm_dest, xmm_src/memory
```

Операция `sqrtps` вычисляет квадратные корни четырех 32-битовых значений, хранящихся в некоем регистре XMM или в 128-разрядном операнде в памяти, и записывает результаты в целевой XMM регистр.

Пример. Вычисление  $\sqrt{x_0}, \sqrt{x_1}, \sqrt{x_2}, \sqrt{x_3}$

```
_asm {
    sqrtps xmm0, x // Вычисление квадратных корней
    movaps z, xmm0 // сохранение результатов в z
}
```

## Логические операции

В SIMD осуществляется поддержка всех возможных логических операций. Основные логические операции **AND**, **OR**, **XOR** производятся с упакованными 32-битовыми значениями с одинарной точностью и имеют следующий формат.

```
andps xmm_dest, xmm_src/memory
orps  xmm_dest, xmm_src/memory
xorps xmm_dest, xmm_src/memory
```

Каждая логическая операция применяется к операнду в памяти или регистру XMM и целевому регистру. Результат всегда записывается в целевой регистр.

Пример. Вычисление  $z = ((x \text{ AND } y) \text{ OR } z)$

```
_asm {
    movaps xmm0, x // Перемещение содержимого x в xmm0
    movaps xmm1, y // Перемещение содержимого y в xmm1
    movaps xmm2, z // Перемещение содержимого z в xmm2

    andps xmm0, xmm1 // xmm0 = x AND y
    orps xmm0, xmm2 // xmm0 = (x AND y) OR z
    movaps z, xmm0 // Сохранение результатов в z
}
```

## Операции сравнения

Наконец, существует множество команд сравнения, однако основная команда `cmprrs` аналогична стандартной команде `cmp` IA-32. Команда `cmprrs` предусматривает сравнение упакованных значений с одинарной точностью и имеет следующий вид.

`cmprrs xmm_dest, xmm_src, immediate_8`

Сравниваются четыре пары упакованных 32-битовых значений и результаты сохраняются в `xmm_dest` в виде четырех 32-битовых строк, состоящих из сплошных нулей или единиц, в зависимости от результатов сравнения. Вид сравнения задается с помощью параметра `immediate_8`. Возможные типы сравнений представлены в табл. 16.4.

**Таблица 16.4. Операторы сравнения**

Биты 0-2	Вид сравнения
0	Равно
1	Меньше
2	Меньшеилиравно
3	Неупорядочены
4	Не равно
5	Не меньше
6	Неменьшеилиравно
7	Упорядочены

Примечание, Биты 4-7 зарезервированы

Пример: Проверка неравенства `x[] < y[]`

```
_asm {
    movaps xmm0, y
    cmprrs xmm0, x // xmm0 содержит результаты сравнения
}
```

## Встроенные функции

Встроенные функции очень похожи на сами команды. Лучше всего внимательно посмотреть файл `xmmintrin.h`, где приводятся все встроенные команды; мы же рассмотрим только основные.

### Арифметические операции

```
_mm128_mm_add_ps(_mm128 a, _mm128 b); // a+b
_mm128_mm_sub_ps(_mm128 a, _mm128 b); // a-b
_mm128_mm_mul_ps(_mm128 a, _mm128 b); // a*b
_mm128_mm_div_ps(_mm128 a, _mm128 b); // a/b
_mm128_mm_sqrt_ps(_mm128 a); // Квадратный корень из a
_mm128_mm_rcp_ps(_mm128 a); // Значение, обратное a
_mm128_mm_rsqrt_ps(_mm128 a); // Значение, обратное
// квадратному корню из a
_mm128_mm_min_ps(_mm128 a, _mm128 b); // min(a,b)
_mm128_mm_max_ps(_mm128 a, _mm128 b); // max(a,b)
```

Пример: `m0=m1+m2`

```
m0 = _mm_add_ps(m1, m2);
```

### Логические операции

```
_mm128_mm_and_ps(_mm128 a, _mm128 b); // a AND b
_mm128_mm_andnot_ps(_mm128 a, _mm128 b); // a NAND b
_mm128_mm_or_ps(_mm128 a, _mm128 b); // a OR b
_mm128_mm_xor_ps(_mm128 a, _mm128 b); // a XOR b
```

Пример: `m0 = m1 AND m2`

```
m0 = _mm_and_ps(m1, m2);
```

### Операции сравнения

```
_mm128_mm_cmpeq_ps(_mm128 a, _mm128 b); // a == b
_mm128_mm_cmplt_ps(_mm128 a, _mm128 b); // a < b
_mm128_mm_cmple_ps(_mm128 a, _mm128 b); // a <= b
_mm128_mm_cmpgt_ps(_mm128 a, _mm128 b); // a > b
_mm128_mm_cmpge_ps(_mm128 a, _mm128 b); // a >= b
_mm128_mm_cmpneq_ps(_mm128 a, _mm128 b); // a != b
_mm128_mm_cmpnlt_ps(_mm128 a, _mm128 b); // !(a < b)
_mm128_mm_cmpnle_ps(_mm128 a, _mm128 b); // !(a <= b)
_mm128_mm_cmpngt_ps(_mm128 a, _mm128 b); // !(a > b)
_mm128_mm_cmpnge_ps(_mm128 a, _mm128 b); // !(a >= b)
_mm128_mm_cmpord_ps(_mm128 a, _mm128 b);
_mm128_mm_cmpunord_ps(_mm128 a, _mm128 b);
```

Пример: `m0 = m1 < m2`

```
m0 = _mm_cmplt_ps(m1, m2);
```

Можно по желанию использовать как встроенные функции, так и ассемблерный код; однако следует понимать, что многие встроенные функции представляют собой несколько отдельных SIMD-команд, поэтому при оптимизации следует проанализировать полученный ассемблерный результат, чтобы проверить, нельзя ли добиться лучших результатов, написав вручную соответствующий SIMD-код в особо важных для производительности фрагментах.

## Где найти дополнительную информацию

Когда я был подростком, мне удавалось бесплатно получать у компаний микросхемы, книги и программное обеспечение на тысячи долларов, достаточно было просто позвонить и попросить. Сегодня это тоже возможно, проблема только в том, чтобы найти нужного человека. Компания Intel предлагает на своем Web-узле множество бесплатных материалов, которые должны вас заинтересовать. Там можно найти полные руководства по

разработке программного обеспечения, программированию и оптимизации на базе Pentium III/4, причем не только в формате PDF, а в виде **настоящих книг**, стоимостью 250 долл. и больше, которые **будут** присланы вам на дом! Все, что от вас требуется, — это прочитать полученные книги, и вы станете настоящим специалистом в сфере оптимизации на базе Pentium. Ниже приводится URL, который позволяет оформить заказ в Литературном **центре** компании Intel (Intel's Literature Center), <http://developer.intel.com/design/pentium4/manuals/index2.htm>

На этом Web-узле вы найдете следующие руководства.

- "IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture"
- "IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference"
- "IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide"
- "Intel Pentium 4 and Intel Xenon Processor Optimization Manual"

Нет ничего лучше, чем получить что-то "на шару"! Кроме того, на узле компании Intel содержится масса статей по оптимизации на базе Pentium. Если вы собираетесь создавать игры, крайне желательно знать этот материал.

## Класс трехмерных векторов с поддержкой SIMD

**Теперь**, когда у вас уже есть минимальные знания по теории SIMD и принципам использования основных математических команд, попытаемся их объединить и создать небольшой класс C++, который обеспечит поддержку трехмерных векторов с компонентом `w` и использует SIMD для выполнения вычислений.

Данный класс ни в коей мере не является **завершенным**, но это хороший способ поэкспериментировать и потренироваться в написании SIMD-кода. Класс поддерживает четырехмерные векторы формата `[x, y, z, w]`, т.е. каждый объект имеет тот же вид, что и отдельный регистр SIMD. В классе реализованы функции сложения, **вычитания**, скалярного **произведения**, определения длины, вывода, доступа к массиву и многие другие вспомогательные функции. Кроме того, многие функции реализованы с использованием условной **компиляции**, что позволяет **использовать** либо ассемблерную версию, либо встроенные функции. Параметры, **управляющие** условной компиляцией, выглядят **следующим** образом.

```
// Тип используемых операций задается путем установки одного
// из этих параметров равным 1, а второго - 0
#define SIMD_INTRINSIC 0
#define SIMD_ASM 1
```

Ниже приводится полное определение класса. Оно снабжено подробными комментариями; постарайтесь прочитать их и проследить за каждой **SIMD-командой**. Единственная операция, которая может вызвать некоторое затруднение, — это скалярное произведение. Дело в том, что при вычислении скалярного произведения легко параллельно умножить координаты двух векторов, но сложно затем просуммировать содержащиеся в ХММ регистре результаты "горизонтально". Для получения конечного результата требуется совместное использование перераспределения (с помощью команды `shufps`) и сложения (`addps`). Итак, вот определение класса.

```
// Новый класс вектора, который поддерживает SIMD SSE.
// Обратите внимание на многочисленные способы доступа к
```

```

// данным-членам, что позволяет обеспечить прозрачную
// поддержку присваиваний, доступа к данным, а также
// использовать внутреннюю библиотеку, не выполняя
// приведения типов

class C_VECTOR4D
{
public:

    union
    {
        _declspec(align(16)) _m128 v; // Тип данных SIMD
        float M[4]; // Объявление массива
        // Явные имена
        struct
        {
            float x,y,z,w;
        }; // struct
    }; // union

    // Примечание. declspec здесь является избыточным, поскольку
    // тип _m128 заставляет компилятор производить 16-байтовое
    // выравнивание данных. Поэтому до тех пор, пока _m128
    // является частью объединения, declspec не является
    // необходимым :) Однако вреда от него не будет, и при
    // определении локальных и глобальных переменных всегда
    // следует помещать declspec(align(16)), чтобы знать
    // наверняка, что данные выровнены по 16-байтовым границам

    // КОНСТРУКТОРЫ //////////////////////////////////////

    C_VECTOR4D(>
    {
        // Конструктор по умолчанию инициализирует вектор
        // значениями 0,0,0,1
        x=y=z=0; w=1.0;
    } // C_VECTOR4D

    //////////////////////////////////////

    C_VECTOR4D(float _x, float _y, float _z, float _w = 1.0)
    {
        // Инициализирует вектор переданными значениями
        x = _x;
        y = _y;
        z = _z;
        w = _w;
    } // C_VECTOR4D

    // ФУНКЦИИ //////////////////////////////////////

    void init(float _x, float _y, float _z, float _w = 1.0)
    {

```

```

// Инициализирует вектор переданными значениями
x = _x;
y = _y;
z = _z;
w = _w;
} // init

////////////////////////////////////

void zero(void)
{
    // Инициализирует вектор значениями 0,0,0,1
    x=y=z=0; w=1.0;
} // zero

////////////////////////////////////

float length(void)
{
    // Вычисляет длину вектора
    C_VECTOR4D vr = *this;

    // Задаем w=0
    vr.w = 0;

    // Версия на языке ассемблера?
    #if (SIMD_ASM==1)

        // Ассемблерная версия скалярного произведения,
        // поскольку его результат нужен для определения длины
        // по формуле length = sqrt(v*v)
        _asm
        {
            // Сначала находим скалярное произведение this*this
            movaps xmm0, vr.v // Помещаем левый операнд в xmm0
            mulps xmm0, xmm0 // Вертикально умножаем операнды

            // В данный момент xmm0 -
            // [(v1.x*v2.x), (v1.y*v2.y), (v1.z*v2.z), (1*1)]
            // Пусть xmm0 - [x,y,z,1] -
            // [(v1.x*v2.x), (v1.y*v2.y), (v1.z*v2.z), (1*1)]
            // Теперь необходимо просуммировать компоненты
            // x,y,z, чтобы получить одну скалярную величину и
            // найти значение скалярного произведения по
            // формуле:  $dp = x + y + z - x^2 + y^2 + z^2$ 

            // Начало
            // xmm0: = [x,y,z,1]
            // (Примечание: регистры показаны от младших к
            // старшим)
            // xmm1: - [?,?,?,?]
            movaps xmm1, xmm0 // Результат копируется в xmm1
            // xmm0: - [x,y,z,1]

```

```

// xmm1: = [x,y,z,1]

shufps xmm1, xmm0, SIMD_SHUFFLE(0x01,0x00,0x03,0x02)
// xmm0: = [x,y,z,1]
// xmm1: = [z,1,x,y]

addps xmm1, xmm0
// xmm0: - [x,y,z,1]
// xmm1: - [x+z,y+1,x+z,y+1]

shufps xmm0, xmm1, SIMD_SHUFFLE(0x02,0x03,0x00,0x01)
// xmm0: = [y,x,y+1,x+z]
// xmm1: = [x+z,y+1,x+z,y+1]

// Наконец, можно выполнить сложение!
addps xmm0, xmm1
// xmm0: - [x+y+z,x+y+1,x+y+z+1,x+y+z+1]
// xmm1: - [x+z,y+1,x+z,y+1]
// xmm0.x содержит скалярное произведение
// xmm0.z, xmm0.w содержат скалярное произведение+1

// Теперь младшее двойное слово содержит скалярное
// произведение. Извлекаем квадратный корень
sqrtss xmm0, xmm0

movaps vr, xmm0 // Сохраняем результаты

} // asm

#endif

// Встроенные функции
#if (SIMD_INTRINSIC==1)
#endif

// Возврат результата
return(vr.x);

} // length

// ПЕРЕГРУЖЕННЫЕ ОПЕРАТОРЫ //////////////////////////////////////

float& operator[](int index)
{
    // Возвращает i-й элемент массива
    return(M[index]);
} // operator[]

////////////////////////////////////

C_VECTOR4D operator+(C_VECTOR4D &v)
{
    // Складывает вектор "this" и переданный вектор

```

```

    __declspec(align(16)) C_VECTOR4D vr;
    // Используется для хранения результата;
    // 16-байтовое выравнивание

    // Версия на языке ассемблера
    #if (SIMD_ASM==1)
    _asm
    {
        mov esi, this          // "this" содержит указатель на
                                // левый операнд
        mov edi, v             // v указывает на правый операнд
        movaps xmm0, [esi]     // esi указывает на первый
                                // вектор, помещаем его в xmm0
        addps xmm0, [edi]      // edi указывает на второй
                                // вектор, прибавляем его к xmm0
        movaps vr, xmm0        // Помещаем результат в итоговый
                                // вектор
    } // asm

tfendif

// Встроенные функции
tfif(SIMD_INTRINSIC—1)

    vr.v = _mm_add_ps(this->v, v.v);

flendif

    // Всегда устанавливаем w=1
    vr.w = 1.0;

    // Возврат результата
    return (vr);
} // operator+

////////////////////////////////////

C_VECTOR4D operator-(C_VECTOR4D &v)
{
    // Вычитает вектор "this" и переданный вектор

    __declspec(align(16)) C_VECTOR4D vr;
    // Используется для хранения результата
    // (16-байтовое выравнивание)

    // Версия на ассемблере
    #if (SIMD_ASM==1)
    _asm
    {
        mov esi, this          // "this" содержит указатель на

```

```

        movedi, v           // левый операнд
        movaps xmm0, [esi]  // v указывает на правый операнд
                             // esi указывает на первый
                             // вектор, помещаем его в xmm0
        subps xmm0, [edi]   // edi указывает на второй
                             // вектор, вычитаем его из xmm0
        movaps vr, xmm0     // Помещаем результат в итоговый
                             // вектор

    } // asm

#endif

// Встроенные функции
#if (SIMD_INTRINSIC==1)

    vr.v = _mm_sub_ps(this->v, v.v);

#endif

// Всегда устанавливаем w=1
vr.w = 1.0;

// Возврат результата
return (vr);

} // operator-

////////////////////////////////////

float operator*(C_VECTOR4D &v)
{
    // Оператор * будет обозначать скалярное произведение,
    // поскольку эта операция встречается чаще. Вычисляется
    // скалярное произведение вектора "this" и переданного
    // вектора

    __declspec(align(16)) C_VECTOR4D vr;
    // Используется для хранения результата
    // (16-байтовое выравнивание)

    // Версия на ассемблере
    #if (SIMD_ASM==1)
        _asm
        {
            mov esi, this           // "this" содержит указатель на
                                    // левый операнд
            movedi, v               // v указывает на правый операнд
            movaps xmm0, [esi]     // Помещаем левый операнд в xmm0
            mulps xmm0, [edi]       // Вертикально перемножаем
                                    // операнды

            // В данный момент, xmm0 =

```

```

// [(v1.x*v2.x), (v1.y*v2.y), (v1.z*v2.z), (1*1)]
// Пусть xmm0 = [x,y,z,1] -
// [(v1.x*v2.x), (v1.y*v2.y), (v1.z*v2.z), (1*1)]
// Необходимо просуммировать компоненты x,y,z, чтобы
// получить скалярную величину для вычисления
// скалярного произведения по формуле:
// dp = x+y+z, где x = x1*x2, y = y1*y2, z = z1*z2

// Начало
// xmm0 = [x,y,z,1]
// (Примечание: регистры располагаются в порядке от
// младших к старшим)
// xmm1 = [?,?,?,?]
movaps xmm1, xmm0 // Результат копируется в xmm1
// xmm0 = [x,y,z,1]
// xmm1 = [x,y,z,1]

shufps xmm1, xmm0, SIMD_SHUFFLE(0x01,0x00,0x03,0x02)
// xmm0 = [x,y,z,1]
// xmm1 = [z,1,x,y]

addps xmm1, xmm0
// xmm0 = [x,y,z,1]
// xmm1 = [x+z,y+1,x+z,y+1]

shufps xmm0, xmm1, SIMD_SHUFFLE(0x02,0x03,0x00,0x01)
// xmm0 = [y,x,y+1,x+z]
// xmm1 = [x+z,y+1,x+z,y+1]

// Наконец, можно выполнить сложение!
addps xmm0, xmm1
// xmm0 = [x+y+z,x+y+1,x+y+z+1,x+y+z+1]
// xmm1 = [x+z,y+1,x+z,y+1]
// xmm0.x содержит скалярное произведение
// xmm0.z, xmm0.w содержит скалярное произведение+1
movapsvr, xmm0
} // asm

```

tfendif

```

// Встроенные функции
#if (SIMD_INTRINSIC==1)
vr.v = _mm_mul_ps(this->v, v.v);
return(vr.x + vr.y + vr.z);

```

tfendif

```

// Возврат результата
return(vr.x);

```

} // operator\*

////////////////////////////////////

```

void print(void)
{
    // Данная функция-член выводит вектор
    printf("\nv - [%f, %f, %f, %f]",
           this->x, this->y, this->z, this->w);
} // print

////////////////////////////////////

}; //class C_VECTOR4D

```

Чуть позже мы рассмотрим несколько простых примеров **использования** класса, но сначала обратите внимание на использование макрокоманды **SIMD\_SHUFFLE()**, которую мы уже обсуждали ранее в разделе о **перемещениях** данных. Данная макрокоманда просто перемешивает **SIMD-слово**, расположенное в определенной позиции источника, в другую **позицию** целевого регистра, что позволяет менять слова местами и упорядочивать их таким образом, чтобы с ними было удобнее работать.

На заключительном этапе вычисления скалярного произведения требуется выполнить **горизонтальное** сложение. После первого этапа вычисления результаты параллельных **SIMD-умножений** выглядят **следующим** образом.

```
[ux*vx, uy*vy, uz*vz, 1*1]
```

Проблема состоит в том, что для получения окончательного результата нам нужно выполнить сложение ( $ux*vx + uy*vy + uz*vz$ ). С помощью обычных **SIMD-команд** это сделать невозможно, поскольку команды для горизонтального сложения не существует. Поэтому для получения конечного результата приходится копировать результаты, сдвигать их и выполнять сложение.

Перейдем к использованию класса. Определим два **SIMD-вектора**

```
C_VECTOR4D v1(1,2,3), v2(4,5,6), result;
```

Сложение выглядит следующим образом.

```
result = v1 + v2;
```

Выведем результат.

```
result.print();
```

Вычислим длину результирующего вектора и выведем ее.

```
length = result.length();
```

```
cout << length;
```

Наконец, вычислим скалярное произведение двух векторов.

```
float dp = v1 * v2;
```

Трудно представить **что-либо** более простое! Чтобы увидеть демонстрационную версию класса в действии, обратимся к **DEMO116\_3.CPP|EXE**. Программа содержит полную версию данного класса и короткую демонстрационную программу, аналогичную предыдущим примерам. Данную демонстрационную программу, как и предыдущую, нужно компилировать как консольное приложение; необходимо также иметь **SSE-совместимую** операционную систему и процессор Pentium III+.

Если вас это интересует, можно попробовать добавить в класс умножение матриц и другие операции. Обдумайте, каким будет самый быстрый способ ввода и вывода для каждой операции. Написание хорошей **SIMD-программы** требует определенных **размыш-**

лений, как и написание класса. Если использовать методы "в лоб", окажется, что она не-намного лучше, чем обычная программа на C++.

## Некоторые оптимизационные приемы

В данном разделе предлагается несколько интересных и полезных оптимизационных приемов; некоторые из них я разработал сам, другие почерпнул из перечисленных в конце главы источников.

### Прием 1. Избавление от `_ftol()`

Если в программе встречается присваивание значения типа `float` переменной типа `int`, компилятор будет многократно вызывать внутреннюю функцию `_ftol()`, что может катастрофически замедлить выполнение программы. Рассмотрим следующий пример.

```
float f= 10.5;
inti = f;
```

Чтобы этого избежать, следует использовать встроенный ассемблер и команды `FPU`, такие как `fistp/fst`.

```
_asm
f
ftdf;
fistp i;
!
```

Можно также использовать флаг компилятора `/QIfist`, что приведет к отключению режима округления, и функция `_ftol()` вызываться не будет.

При этом следует остерегаться побочных эффектов и потери точности. Испытайте эти приемы на отдельных модулях. Вызов функции `_ftol()` является одним из самых узких мест в отношении производительности: вы думаете, что пишете оптимальный код, просто присваиваете в нем переменной типа `int` значение типа `float`, и без вашего ведома производится вызов функции!

### Прием 2. Задание управляющего слова `FPU`

Процессор для работы с плавающей точкой содержит управляющее слово, которое определяет режим округления, точность и денормализацию. С управляющим словом `FPU` можно работать с помощью языка ассемблера; однако в C/C++ есть функция `_control87()`, которая тоже позволяет это делать.

```
unsigned int _control87(unsigned int control,
unsigned int mask);
```

В табл. 16.5 представлены некоторые наиболее часто используемые управляющие флаги, которыми можно управлять в целях повышения производительности.

Например, при выполнении вычислений с очень маленькими числами могут возникать исключительные ситуации. Если отключить контроль денормализации, обработка ускорится. Кроме того, можно понизить точность вычислений и посмотреть, будет ли все работать по-прежнему. Ниже приводится код установки обоих описанных свойств.

```
// Установка одиночной точности
_control87(_PC_24, _MCW_PC);
```

```
// Отключение контроля денормализации
_control87(_DN_FLUSH, _MCW_DN);
```

**Таблица 16.5. Параметры функции \_control87()**

Маска	Шестнадцатеричное значение	Управляющая константа	Шестнадцатеричное значение
_MCW_DN (управление денормализацией)	0x03000000	_DN_SAVE	0x00000000
		_DN_FLUSH	0x01000000
_MCW_RC (управление округлением)	0x00000300	_RC_CHOP	0x00000300
		_RC_UP	0x00000200
		_RC_DOWN	0x00000100
		_RC_NEAR	0x00000000
_MCW_PC (управление точностью)	0x00030000	_PC_24 (24 бита)	0x00020000
		_PC_53 (53 бита)	0x00010000
		_PC_64 (64 бита)	0x00000000

### Прием 3. Быстрое обнуление значений с плавающей точкой

Двоичное кодирование целых значений и значений с плавающей точкой различно, т.е. в двоичном представлении `(int)1` не равно `(float)1`. Однако `(int)0` оказывается равным `(float)0`, поэтому если вам нужно быстро очистить большую область памяти и заполнить ее нулевыми значениями с плавающей точкой, можно заменить фрагмент кода

```
float x[1000];
```

```
for (inti=0; i < 1000; i++)x[i] = 0;
```

следующим кодом:

```
memset(x, 0x0, sizeof(float)*1000);
```

Можно также использовать язык ассемблера и заполнение нулями 32-битовых слов,

```
_asm
i
mov edi, x ; edi указывает место назначения в памяти
mov ecx, 1000/4 ; Число перемещаемых 32-битовых слов
mov eax, 0 ; 32-разрядные данные
rep stosd ; Перемещение данных
} // asm
```

## Прием 4. Быстрое извлечение квадратного корня

Ниже предлагается быстрый алгоритм извлечения квадратного корня, который вычисляет квадратный корень переданного числа с ошибкой в пределах 5%.

```
float Fast_SquareRT(float f)
{
    float result; // Используется для передачи результата
    __asm
    {
        moveax, f
        sub eax, 0x3f800000
        sar eax, 1
        add eax, 0x3f800000
        mov result, eax
    } // asm

    // Возврат результата
    return(result);
} // Fast_SquareRT
```

## Прием 5. Линейно-кусочный арктангенс

Мы уже использовали таблицы поиска и другие специальные приемы для вычисления арктангенса. Однако все наши методы охватывали некие частные случаи. Существует интересное точное решение данной задачи, которое позволяет для любых  $(dx, dy)$  вычислить арктангенс с очень маленькой погрешностью.

На первом этапе данный алгоритм использует дерево решений, чтобы выяснить, в каком октанте находится  $(dx, dy)$ , а затем с помощью некоторой функции вывода представляет окончательный результат в диапазоне  $[0, 8]$ , который можно преобразовать в любой требуемый формат, например  $0-360^\circ$  или  $0-2\pi$  радиан. На рис. 16.26 представлена графическая иллюстрация алгоритма. Остается просто закодировать соответствующие условия для каждой восьмой части круга и вывод полученных значений.

## Прием 6. Увеличение указателя

При осуществлении доступа к массивам посредством указателей не следует поступать так, как показано ниже.

```
int *p = base;
```

```
*p = 5;
p++;
```

```
*p = 6;
p++;
```

```
*p = 7;
```

Вместо этого лучше использовать следующий код.

```
p[0] = 5;
p[1] = 6;
p[2] = 7;
```

Доступ к данным с использованием элементов массива производится быстрее, чем с помощью инкрементного приращения указателя.

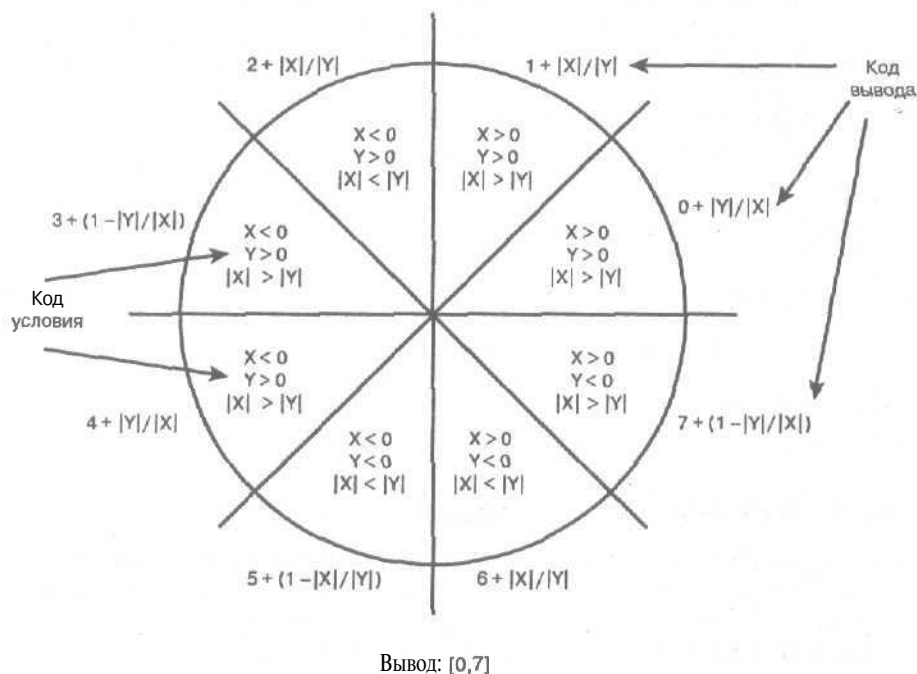


Рис. 16.26. Графическая иллюстрация алгоритма приближенного вычисления арктангенса

## Прием 7. Вынесение if из циклов

Казалось бы, это элементарное требование здравого смысла, однако данная ситуация постоянно встречается в программах, так что я счел нужным упомянуть ее. Предположим, у нас есть некий цикл и внутри цикла на каждой  $n$ -й итерации необходимо выполнить определенную операцию. Пишется следующий код.

```
for (int i=0; i < 256; i++)
{
    // проверка специального случая
    if ((i % 64) == 0)
    {
        // код обработки специального случая
    }
    else
    {
        // код обработки стандартного случая
    } // else
} // for
```

Это ужасно! Почему просто не разбить данный цикл на четыре цикла, подобных приведенным ниже? Тогда оператор if не понадобится вовсе.

// Код обработки специального случая для  $i \equiv 0$

```
// Цикл для значений 1-63
for (i = 1; i < 64; i++)
    // Код обработки стандартного случая

// Код обработки специального случая для i=64

// Цикл для значений 65 - 127
for (; i < 128; i++)
    // Код обработки стандартного случая

// Код обработки специального случая для i=128

// Цикл для значений 129 - 192
for (; i < 192; i++)
    // Код обработки стандартного случая

// Код обработки специального случая для i=192

// Цикл для значений 193 - 255
for (; i < 256; i++)
    // Код обработки стандартного случая
```

## Прием 8. Ветвление конвейера

Процессоры Pentium III/4 имеют большую глубину конвейерных вычислений (до 20 инструкций), и при неправильном прогнозировании ветвления отключается весь конвейер целиком, так что неправильного прогнозирования нужно избегать. Кроме того, при выполнении программы процессор обычно выполняет код, который, как он считает, потребуется на следующем шаге. Таким образом, неправильное прогнозирование не только останавливает конвейер, но и приводит к обработке кода, который никогда не понадобится! Для избежания неправильного прогнозирования, следует так составлять программу, чтобы: в большинстве случаев условный оператор if выполнялся и только изредка условие оказывалось ложным.

## Прием 9. Выравнивание данных

Процессор Pentium лучше всего работает с 16-/32-байтовым выравниванием, поэтому, по возможности, следует выравнивать все структуры данных на указанные границы и/или так заполнять структуры данных, чтобы они были кратны этим числам. Выравниванием можно управлять как с помощью опций компилятора, так и с помощью описания `__declspec(align(value))`.

## Прием 10. Все короткие функции нужно сделать встраиваемыми

Все часто вызываемые функции, состоящие из 10–20 строк кода, следует сделать встраиваемыми (inline). Для этого необходимо переместить ее из исходного файла C/C++ в заголовочный файл, поскольку компилятору нужен исходный текст функции, а не только ее прототип. Приведу пример. В ходе профилирования я обнаружил, что функции скалярного произведения векторов и умножения матриц вызываются гораздо чаще, чем другие математические функции. Сделав эти функции встраиваемыми, мне удалось повысить производительность на 3-5% — для этого понадобилось единственное ключевое слово!

## Список литературы

- *The Graphics Gems Series*, Academic Press
- *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Press
- *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Press
- *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Press
- *Intel Pentium 4 and Intel Xeon Processor Optimization Manual*. Intel Press
- Richard Gerber. *The Software Optimization Cookbook*. Intel Press

## Резюме

В данной главе мы почти не рассматривали оптимизацию на низком уровне, уделив основное внимание знакомству со средствами и методами, которые вы, возможно, никогда не использовали. Сложно передать, насколько важно для вас правильно воспринять все обсуждавшиеся технологии — они могут коренным образом изменить ваше представление о программировании процессоров следующего поколения и игровых приложений. Давно прошли те времена, когда для оптимизации кода использовали секундомеры, и теперь вы просто обязаны использовать все достижения современной технологии.



# Предметный указатель

## I

I/z-буферизация, 1078

## A

Ammeraal, Leendert, 52?

## B

BHV, 1237

Октадеревья, 1242; 1255

BSP, 1148

Конвейер визуализации, 1191

Обход дерева, 1156; 1177

Отбраковка, 1208

Построение дерева, 1161

## C

Catmull, Edwin, 961

COM, 110–16

## D

Direct3DIM, 109

Direct3DRM, 109

DirectAudio, 188

DirectDraw, 108

DirectInput, 109; 183

DirectMusic, 109; 188

DirectPlay, 109

DirectShow, 110

DirectSound, 108; 188

DirectSound3D, 108

DirectX, 25; 106–10

DirectXAudio, 110

DirectX Graphics, 110

## K

Eberly, David H., 1356

Edelstein, Paul, 512

## G

Gauss, Karl, 295

GDI, 91

## H

Haines, Eric, 488

HAL, 107

Hamilton, William Rowan, 278

HEL, 107

## I

Intel C++, 1372

## L

Leibnitz, Gottfried Wilhelm, 291

## M

Mellor, Michael, 1337

MMX, 1377

## N

Newton, Isaac, 291

## P

PVS, 1226

Порталы, 1232

Представление, 1230

## R

Roehl, Bernie, 485

Romero, John, 1352

Rule, Keith, 501

## S

Schneider, Philip J., 1356

Sedgewick, Robert, 523

SIMD, 872; 1377

## U

UVN-камера, 554

## V

Varanese, Alex, 1352  
Visual C++, 45  
VTune, 1365

## W

Win32, 80  
Win32 API  
    CreateWindowEx(), 93  
    DispatchMessage(), 101  
    GetMessage(), 100  
    GetStockObject(), 91  
    LoadCursor(), 91  
    LoadIcon(), 90  
    MessageBeep(), 86  
    MessageBox(), 81; 84  
    PeekMessage(), 104  
    PostQuitMessage(), 100  
    RegisterClassEx(), 92  
    SystemParametersInfo(), 210  
    TranslateMessage(), 101  
    WinMain(), 81  
WinProc, 95  
Wolfram, Stephen, 858  
Woodrell, Joe, 1314

## Z

Z-буферизация, 961; 1032  
    И альфа-смешивание, 1049  
    Оптимизация, 998  
    Реализация, 975  
    Смешанная, 1001  
Z-сортировка, 959

## A

Аксонметрическое преобразование, 448; 585  
Алгоритм  
    Брезенхама, 805  
Алгоритм Невеля-Санча, 959  
Алгоритм художника, 686; 714; 958  
Альфа-смешивание, 636; 1043  
    Z-буферизация, 1048  
    Ландшафты, 1065  
    Оптимизация, 1043  
Аммераал, Линдерт, 523  
Анимация, 128; 412; 760

## Б

Базис, 262  
Бесконечность, 291  
Бесконечные ряды, 296  
Билинейная интерполяция, 853  
Билинейная фильтрация текстуры, 1103  
Блиттер, 174  
Блокировка памяти, 124  
Блокировка подвески, 439  
Брезенхама алгоритм, 652  
Буфер кадра, 122

## В

Варанес, Алекс, 1352  
Вектор, 238-47  
    Базис, 262  
    Векторное умножение, 245  
    Вычитание, 241  
    Длина, 239  
    Норма, 239  
    Нормализация, 240  
    Орт, 262  
    Скалярное произведение, 242  
    Сложение, 241  
    Умножение на скаляр, 240  
Видимость объекта, 714  
Внешняя поверхность, 473  
Воксельная графика, 513  
Вольфрам, Стивен, 858  
Вудрелл, Джо, 1314

## Г

Гамильтон, Вильям, 278  
Гамма-коррекция, 1122  
Гаусс, Карл, 295  
Гиперплоскость, 1153  
Глобальные переменные, 37  
Графический контекст, 100

## Д

Декартовы координаты, 223; 229  
Дескриптор, 90  
Дискретизация, 850  
Дифференциальное исчисление, 291-308

## З

Затенение  
    плоское, 687  
    по Гуро, 710; 791; 823; 1034

по Фонгу, 640; 713  
Зеркальное отражение, 639

## И

Игровой мир, 481  
Идентификатор интерфейса, 113  
Инструментарий трехмерного моделирования, 483  
Интеграл, 303  
Интерфейс, 111  
Искусственный интеллект, 420

## К

Карта освещения, 1296  
Касательная, 300  
Каскадные таблицы соответствия, 870  
Кватернион, 278–90; 320; 335  
    Вычитание, 284  
    Мультипликативный обратный, 286  
    Сложение, 284  
    Сопряженный, 285  
    Умножение, 284  
Кисть, 91  
Класс Windows, 87  
    Регистрация, 92  
Комплексное число  
    Норма, 282  
Комплексные числа, 278  
Конвейер визуализации, 423  
Конечные ряды, 294  
Контекст визуализации, 1018; 1135  
Контекст устройства, 89  
Кэтмулл, Эдвин, 961  
Кэширование вычислений, 844  
Кэширование поверхности, 1299

## Л

Ландшафт, 904; 935; 1065  
    Генерация, 936  
    Использование текстуры, 939  
Лейбниц, Готтфрид, 291

## М

Масштабирование, 258  
Математические обозначения, 222  
Матрица, 247–63; 317; 333  
    Детерминант, 253  
    Единичная, 249  
    Квадратная, 249  
    Масштабирование точки, 258  
    Матричное умножение, 251

Мультипликативная обратная, 252  
Нулевая, 249  
Определитель, 253  
Ортонормированная, 262  
Перенос точки, 257  
Поворот точки, 259  
Скалярное умножение, 250  
Сложение, 250  
Транспонированная, 250  
Меллор, Майкл, 1337  
Метод исключения Гаусса, 253  
Мнимая единица, 278  
Многопроходная визуализация, 1133  
Многоугольник, 267  
Множественное отображение текстур, 1108  
Морфинг, 506  
Муар, 1109

## Н

Направленное освещение, 643  
Начало координат, 223  
Ньютон, Исаак, 291

## О

Область обзора, 431  
Обнаружение столкновений, 1354  
Обработчик событий, 95  
Обработчик сообщений, 100  
Общий свет, 637  
Объемная визуализация, 513  
Ограничивающие иерархические объемы. См. *BHV*  
Однородные координаты, 256  
Октадерево, 1255  
Определение видимых поверхностей, 958  
Освещение по Фонгу, 640  
Ось абсцисс, 223  
Ось ординат, 223  
Отбраковка задних поверхностей, 441  
Отбраковка объектов, 574  
Отбраковка скрытых объектов, 441  
Отображение освещения, 1296  
Отображение фотонов, 632  
Отсечение, 814  
    Алгоритм Вейлера-Азертонна, 897  
    Алгоритм Кохена-Сазерленда, 891  
    Алгоритм Сайруса-Бека, 893  
    Обзор алгоритмов, 884  
Отсечение в пространстве изображения, 455  
Отсечение в пространстве объекта, 455  
Отслеживание лучей, 631

## П

Параметрические уравнения, 271  
Первообразная функция, 304  
Перегрузка функций, 368  
Перенос, 257  
Перспектива, 1069  
Плоское затенение, 687  
Плюскость, 267  
Поворот, 259  
Полутень, 647; 1264  
Полярные координаты, 225; 322  
Порталы, 1234  
Потенциально видимые множества. См. PVS  
Правило Крамера, 253  
Правило левой руки, 473  
Предел, 293  
Прозрачность, 1043; 1273  
Производная, 297  
Профилирование, 1360  
Процедура Windows, 95  
Прямая линия, 263

## Р

Разбиение пространства, 1144  
Двоичное, 1148  
Рал, Кейт, 501  
Рассеянный свет, 637  
Растрезизация треугольников, 805  
Рол, Берни, 485  
Ромеро, Джон, 1352

## С

Световое пятно, 646  
Седжвик, Роберт, 523  
Секущая, 298  
Система координат  
Аксонетрическая, 447  
Двумерная декартова, 223  
Двумерная полярная, 225  
Камеры, 430  
Левая и правая, 229  
Локальная, 424  
Мировая, 427  
Трехмерная декартова, 229  
Трехмерная сферическая, 233  
Трехмерная цилиндрическая, 230  
Экранная, 458  
Скин, 743  
Соглашение о заполнении, 807; 810  
Сообщения Windows, 95; 96  
WM\_CREATE, 97  
WM\_DESTROY, 97

WM\_PAINT, 97  
WM\_QUIT, 105  
Обработчик, 100  
Сопроцессор, 390  
Сопряженные числа, 281  
Список вершин, 471  
Спрайт, 174  
Стиль окна, 88  
Структура  
BITMAP\_FILE, 142  
BITMAP\_IMAGE, 143  
BLINKER, 144  
BOB, 143; 174  
BSPNODEV1, 1160  
CAM4DV1, 552  
CYLINDRICAL3D, 323  
DIJOYSTATE, 183; 216  
DIMOUSESTATE, 183  
DMUSIC\_MIDI, 190  
LIGHTV1, 665  
LIGHTV2, 929  
MATRIX1X2, 145; 319  
MATRIX1X3, 145; 318  
MATRIX1X4, 318  
MATRIX2X2, 319  
MATRIX3X2, 145; 319  
MATRIX3X3, 144; 318  
MATRIX4X3, 317  
MATRIX4X4, 317  
MATV1, 661  
MD2\_ANIMATION, 1345  
MD2\_POLY, 1321  
MSG, 101  
OBJECT4DV1, 478; 520; 691  
OBJECT4DV2, 768  
PAINTSTRUCT, 98  
PALETTEENTRY, 170; 199  
PARMLINE2D, 314; 365  
PARMLINE3D, 315; 366  
pcm\_sound, 189  
PLANE3D, 316; 369  
POINT2D, 313  
POINT2DI, 756  
POINT3D, 313  
POINT3DI, 756  
POINT4D, 314  
POINT4DI, 756  
POLAR2D, 322  
POLY4DV1, 475; 577; 691  
POLY4DV2, 763  
POLYF4DV1, 475; 518  
POLYF4DV2, 766  
POLYF4DV2Q, 1160  
POLYGON2D, 144  
QUAT, 320; 374  
RECT, 99

RENDERCONTEXTV1, 1019; 1137  
 RENDERLIST4DV1, 522; 717  
 RGBAV1, 1058  
 RGBV1, 661  
 SPHERICAL3D, 323  
 VECTOR2D, 313  
 VECTOR2DI, 756  
 VECTOR3D, 313  
 VECTOR3DI, 756  
 VECTOR4D, 314  
 VECTOR4DI, 756  
 VERTEX2DF, 144; 314  
 VERTEX2DI, 144; 314  
 VERTEX4DTV1, 762  
 WNDCLASSEX, 87  
 ZBUFFERV1, 976

Сферические координаты, 233; 323  
 Сценарии, 1352

## T

Таблица поиска цветов, 126  
 Таблицы соответствия, 870  
 Текстура, 799; 1069  
   Билинейная фильтрация, 1103  
   Множественное отображение, 1108  
   С линейно-кусочной перспективой, 1090  
   С точной перспективой, 1086  
   Трилинейная фильтрация, 1132  
 Тень, 647; 1264  
   Масштабирование, 1279  
   Отслеживание источника света, 1283  
   Простая, 1276  
   С учетом вида объекта, 1290  
 Тестирование точки, 886  
 Точечный источник света, 644  
 Трассировка лучей, 5/2  
 Тригонометрические тождества, 237

## Y

Удаление обратных поверхностей, 579

## Φ

Формат

3D Studio Max, 728  
 ASC, 728  
 BioVision, 416  
 BVH, 416  
 COB, 499; 732; 759; 798; 1058  
 DXF, 492  
 MD2, 743; 760; 1312  
 NFF, 488  
 PLG, 485; 758

Загрузчик, 528  
 PLX, 526; 758  
 Загрузчик, 525  
 Quake II, 743  
 SCN, 501  
 trueSpace, 732

Форматное отношение, 452

Функция

\*Extract\_Filename\_From\_Path(), 772  
 \_control87(), 1400  
 Animate\_BOB(), 181  
 Animate\_MD2(), 1346  
 Blink\_Colors(), 172  
 Build\_CAM4DV1\_Matrix\_Euler(), 564  
 Build\_CAM4DV1\_Matrix\_UVN(), 567  
 Camera\_To\_Perspective\_OBJECT4DV2(), 774  
 Camera\_To\_Perspective\_RENDERLIST4DV2(), 774  
 Camera\_To\_Perspective\_Screen\_OBJECT4DV2(), 774  
 Camera\_To\_Perspective\_Screen\_RENDERLIST4DV2(), 774  
 ceil(), 807  
 Clip\_Line(), 160  
 Clip\_Polys\_RENDERLIST4DV2(), 914  
 Close\_Error\_File(), 164  
 Collision\_BOBS(), 182  
 Collision\_Test(), 173  
 Color\_Scan(), 173  
 Compare\_AvgZ\_POLYF4DV2(), 778  
 Compare\_FarZ\_POLYF4DV2(), 778  
 Compare\_NearZ\_POLYF4DV2(), 778  
 Compute\_OBJECT4DV2\_Poly\_Normals(), 775  
 Compute\_OBJECT4DV2\_Radius(), 776  
 Compute\_OBJECT4DV2\_Vertex\_Normals(), 776  
 Compute\_Parm\_Line2D(), 366  
 Compute\_Parm\_Line3D(), 369  
 Compute\_Point\_In\_Plane3D(), 371  
 Copy\_Bitmap(), 169  
 Create\_Bitmap(), 166  
 Create\_BOB(), 176  
 Cull\_OBJECT4DV1(), 575  
 Cull\_OBJECT4DV2(), 77?  
 CYLINDRICAL3D\_To\_POINT3D(), 343  
 CYLINDRICAL3D\_To\_RectXYZ(), 344  
 DDraw\_Attach\_Clipper(), 153  
 DDraw\_Create\_Surface(), 154  
 DDraw\_Fill\_Surface(), 155  
 DDraw\_Flip(), 154; 200  
 DDraw\_Init(), 153  
 DDraw\_Lock\_Back\_Surface(), 156  
 DDraw\_Lock\_Primary\_Surface(), 156  
 DDraw\_Lock\_Surface(), 155  
 DDraw\_Shutdown(), 153  
 DDraw\_Unlock\_Back\_Surface(), 156

DDraw\_Unlock\_Primary\_Surface(), 156  
 DDraw\_Unlock\_Surface(), 155  
 DDraw\_Wait\_For\_Vsync(), 154  
 Destroy\_Bitmap(), 166  
 Destroy\_BOB(), 177  
 Destroy\_OBJECT4DV2(), 772  
 DInput\_Init(), 184  
 DInput\_Init\_Joystick(), 185  
 DInput\_Init\_Keyboard(), 184  
 DInput\_Init\_Mouse(), 184  
 DInput\_Read\_Joystick(), 187  
 DInput\_Read\_Keyboard(), 186  
 DInput\_Read\_Mouse(), 187  
 DInput\_Release\_Joystick(), 185  
 DInput\_Release\_Keyboard(), 185  
 DInput\_Release\_Mouse(), 185  
 DInput\_Shutdown(), 184  
 DMusic\_Delete\_All\_MIDI(), 196  
 DMusic\_Delete\_MIDI(), 195  
 DMusic\_Init(), 195  
 DMusic\_Load\_MIDI(), 195  
 DMusic\_Play(), 196  
 DMusic\_Shutdown(), 195  
 DMusic\_Status(), 196  
 DMusic\_Stop(), 196  
 Draw\_\*\_Tri2\*(), 779  
 Draw\_Bitmap\*(), 168  
 Draw\_BOB\*(), 177  
 Draw\_Clip\_Line\*(), 159  
 Draw\_Filled\_Polygon2D\*(), 158  
 Draw\_Gouraud\_Triangle\*(), 779  
 Draw\_Gouraud\_Triangle2\_16(), 1028  
 Draw\_Gouraud\_Triangle2DWTZB\_16(), 1222  
 Draw\_Gouraud\_TriangleINVZB\_16(), 1083  
 Draw\_Gouraud\_TriangleINVZB\_Alpha16(), 1084  
 Draw\_Gouraud\_TriangleWTZB2\_16(), 1222  
 Draw\_Gouraud\_TriangleZB\_Alpha16(), 1054  
 Draw\_Gouraud\_TriangleZB16(), 986  
 Draw\_Gouraud\_TriangleZB2\_16(), 1033  
 Draw\_Line\*(), 160  
 Draw\_OBJECT4DV1\_Solid\*(), 655  
 Draw\_OBJECT4DV2\_Wire\*(), 775  
 Draw\_Pixel\*(), 161  
 Draw\_QuadFP\_2D(), 157  
 Draw\_Rectangle(), 161  
 Draw\_RENDERLIST4DV1\_Solid\*(), 655  
 Draw\_RENDERLIST4DV2\_Solid\*(), 776  
 Draw\_RENDERLIST4DV2\_Solid\_Alpha16(), 1053  
 Draw\_RENDERLIST4DV2\_Solid16(), 980  
 Draw\_RENDERLIST4DV2\_Solid2\_16(), 1029  
 Draw\_RENDERLIST4DV2\_SolidINVZB\_16(), 1083  
 Draw\_RENDERLIST4DV2\_SolidINVZB\_Alpha16(), 1085  
 Draw\_RENDERLIST4DV2\_SolidZB\_Alpha16(), 1056  
 Draw\_RENDERLIST4DV2\_SolidZB16(), 994  
 Draw\_RENDERLIST4DV2\_SolidZB2\_16(), 1034  
 Draw\_RENDERLIST4DV2\_Wire\*(), 775  
 Draw\_Scaled\_BOB\*(), 178  
 Draw\_Text\_GDI(), 163  
 Draw\_Textured\_Perspective\_Triangle\_FSINVZB\_16(), 1090  
 Draw\_Textured\_Perspective\_Triangle\_FSINVZB\_Alpha16(), 1090  
 Draw\_Textured\_Perspective\_Triangle\_INVZB\_16(), 1090  
 Draw\_Textured\_Perspective\_Triangle\_INVZB\_Alpha16(), 1089  
 Draw\_Textured\_PerspectiveLP\_Triangle\_FSINVZB\_16(), 1095  
 Draw\_Textured\_PerspectiveLP\_Triangle\_FSINVZB\_Alpha16(), 1096  
 Draw\_Textured\_PerspectiveLP\_Triangle\_INVZB\_16(), 1096  
 Draw\_Textured\_PerspectiveLP\_Triangle\_INVZB\_Alpha16(), 1096  
 Draw\_Textured\_Triangle\*(), 778; 858  
 Draw\_Textured\_Triangle\_Alpha16(), 1052  
 Draw\_Textured\_Triangle2\_16(), 1028  
 Draw\_Textured\_TriangleFS\*(), 778; 862  
 Draw\_Textured\_TriangleFS\_Alpha16(), 1052  
 Draw\_Textured\_TriangleFS2\_16(), 1028  
 Draw\_Textured\_TriangleFSINVZB\_16(), 1083  
 Draw\_Textured\_TriangleFSINVZB\_Alpha16(), 1084  
 Draw\_Textured\_TriangleFSWTZB\_16(), 1223  
 Draw\_Textured\_TriangleFSZB\_Alpha16(), 1054  
 Draw\_Textured\_TriangleFSZB16(), 991  
 Draw\_Textured\_TriangleFSZB2\_16(), 1033  
 Draw\_Textured\_TriangleGS\_Alpha16(), 1052  
 Draw\_Textured\_TriangleGSINVZB\_16(), 1083  
 Draw\_Textured\_TriangleGSINVZB\_Alpha16(), 1085  
 Draw\_Textured\_TriangleGSWTZB\_16(), 1223  
 Draw\_Textured\_TriangleGSZB\_Alpha16(), 1054  
 Draw\_Textured\_TriangleINVZB\_16(), 1083  
 Draw\_Textured\_TriangleINVZB\_Alpha16(), 1084  
 Draw\_Textured\_TriangleWTZB\_16(), 1223  
 Draw\_Textured\_TriangleZB\_Alpha16(), 1054  
 Draw\_Textured\_TriangleZB16(), 989  
 Draw\_Textured\_TriangleZB2\_16(), 1033  
 Draw\_Triangle\_2D(), 654  
 Draw\_Triangle\_2D\*(), 157  
 Draw\_Triangle\_2D\_Alpha16(), 1051

Draw\_Triangle\_2D16(), 655  
 Draw\_Triangle\_2D2\*(), 780  
 Draw\_Triangle\_2D3\_16(), 1027  
 Draw\_Triangle\_2DINVZB\_16(), 1078; 1082  
 Draw\_Triangle\_2DINVZB\_Alpha16(), 1084  
 Draw\_Triangle\_2DZB\_Alpha16(), 1054  
 Draw\_Triangle\_2DZB16(), 983  
 Draw\_Triangle\_2DZB2\_16(), 1033  
 Draw\_TriangleFP\_2D(), 654  
 DSound\_Delete\_All\_Sounds(), 193  
 DSound\_Delete\_Sound(), 193  
 DSound\_Init(), 191  
 DSound\_Load\_WAV(), 191  
 DSound\_Play\_Sound(), 192  
 DSound\_Replicate\_Sound(), 192  
 DSound\_Set\_Sound\_Freq(), 194  
 DSound\_Set\_Sound\_Pan(), 194  
 DSound\_Set\_Sound\_Volume(), 193  
 DSound\_Shutdown(), 191  
 DSound\_Status\_Sound(), 193  
 DSound\_Stop\_Sound(), 192  
 EulerZYX\_To\_QUAT(), 376  
 Fast\_Cos(), 342  
 Fast\_Distance(), 163  
 Fast\_Distance\_3D(), 164  
 Fast\_Sin(), 342  
 Find\_Bounding\_Box\_Poly2D(), 164  
 FIXP16\_DIV(), 386  
 FIXP16\_MUL(), 385  
 FIXP16\_Print(), 387  
 Flip\_Bitmap(), 168  
 floor(), 807  
 Generate\_Terrain\_OBJECT4DV2(), 940  
 Get\_Clock(), 172  
 Get\_Palette\_Entry(), 170  
 Hide\_BOB(), 181  
 HLine\*(), 161  
 Init\_CAM4DV1(), 559  
 Init\_OBJECT4DV2(), 772  
 Init\_Parm\_Line2D(), 366  
 Init\_Parm\_Line3D(), 369  
 Insert\_OBJECT4DV2\_RENDERLIST4DV2(), 775  
 Insert\_POLY4DV2\_RENDERLIST4DV2(), 775  
 Insert\_POLYF4DV2\_RENDERLIST4DV2(), 775  
 Intersect\_Parm\_Line3D\_Plane3D(), 372  
 Intersect\_Parm\_Lines2D(), 367; 368  
 Light\_OBJECT4DV2\_World\*(), 777  
 Light\_OBJECT4DV2\_World2\*(), 933  
 Light\_RENDERLIST4DV2\_World\*(), 777  
 Light\_RENDERLIST4DV2\_World2\*(), 933  
 Load\_Animation\_BOB(), 179  
 Load\_Bitmap\_File(), 165  
 Load\_Bitmap\_File2(), 780  
 Load\_Bitmap\_PCX\_File(), 780  
 Load\_Frame\_BOB\*(), 178  
 Load\_Image\_Bitmap\*(), 167  
 Load\_Object\_MD2(), 1331  
 Load\_OBJECT4DV2\_3DSASC(), 776  
 Load\_OBJECT4DV2\_COB(), 777  
 Load\_OBJECT4DV2\_PLG(), 776  
 Load\_Palette\_From\_File(), 171  
 Mat\_Add\_2X2(), 354  
 Mat\_Add\_3X3(), 358  
 Mat\_Add\_4X4(), 361  
 Mat\_Det\_2X2(), 356  
 Mat\_Det\_3X3(), 359  
 Mat\_Init\_2X2(), 354  
 Mat\_Init\_3X2(), 357  
 Mat\_Init\_3X3(), 357  
 Mat\_Init\_4X4(), 360  
 Mat\_Inverse\_2X2(), 356  
 Mat\_Inverse\_3X3(), 359  
 Mat\_Inverse\_4X4(), 364  
 Mat\_Mul\_1X2\_3X2(), 355  
 Mat\_Mul\_1X3\_3X3(), 358  
 Mat\_Mul\_1X4\_4X4(), 362  
 Mat\_Mul\_2X2(), 354  
 Mat\_Mul\_3X3(), 359  
 Mat\_Mul\_4X4(), 361  
 Mat\_Mul\_VECTOR3D\_3X3(), 358  
 Mat\_Mul\_VECTOR3D\_4X3(), 363  
 Mat\_Mul\_VECTOR3D\_4X4(), 362  
 Mat\_Mul\_VECTOR4D\_4X3(), 364  
 Mat\_Mul\_VECTOR4D\_4X4(), 363  
 Model\_To\_World\_OBJECT4DV2(), 775  
 Move\_BOB(), 181  
 Open\_Error\_File(), 164  
 Perspective\_To\_Screen\_OBJECT4DV2(), 775  
 Perspective\_To\_Screen\_RENDERLIST4DV2(), 774  
 PLANE3D\_Init(), 370  
 POINT2D\_To\_POLAR2D(), 343  
 POINT2D\_To\_PolarRTh(), 343  
 POINT3D\_To\_CYLINDRICAL3D(), 344  
 POINT3D\_To\_CylindricalRThZ(), 344  
 POINT3D\_To\_SPHERICAL3D(), 345  
 POINT3D\_To\_SphericalPThPh(), 345  
 POLAR2D\_To\_POINT2D(), 342  
 POLAR2D\_To\_RectXY(), 343  
 Print\_Mat\_2X2(), 357  
 Print\_Mat\_3X3(), 360  
 Print\_Mat\_4X4(), 365  
 QUAT\_Add(), 377  
 QUAT\_Conjugate(), 377  
 QUAT\_Inverse(), 380  
 QUAT\_Mul(), 381  
 QUAT\_Norm(), 378  
 QUAT\_Norm2(), 378

QUAT\_Normalize(), 378  
 QUAT\_Print(), 382  
 QUAT\_Scale(), 378  
 QUAT\_Sub(), 377  
 QUAT\_To\_VECTOR3D\_Theta(), 377  
 QUAT\_Triple\_Product(), 381  
 QUAT\_Unit\_Inverse(), 379  
 Remove\_Backfaces\_OBJECT4DV1(), 580  
 Remove\_Backfaces\_OBJECT4DV2(), 773  
 Remove\_Backfaces\_RENDERLIST4DV1(), 582  
 Remove\_Backfaces\_RENDERLIST4DV2(), 77?  
 Reset\_OBJECT4DV2(), 775  
 Reset\_RENDERLIST4DV2(), 777  
 Rotate\_Colors(), 172  
 Rotate\_Polygon2D(), 158  
 Rotate\_XYZ\_OBJECT4DV2(), 773  
 Save\_Palette(), 171  
 Save\_Palette\_To\_File(), 171  
 Scale\_OBJECT4DV2(), 77?  
 Scale\_Polygon2D(), 159  
 Screen\_Transition(), 162  
 Scroll\_Bitmap(), 169  
 Set\_Anim\_Speed(), 180  
 Set\_Animation\_BOB(), 181  
 Set\_OBJECT4DV2\_Frame(), 772  
 Set\_Palette(), 171  
 Set\_Palette\_Entry(), 170  
 Set\_Pos\_BOB(), 180  
 Set\_Vel\_BOB(), 180  
 Show\_BOB(), 181  
 Solve\_2X2\_System(), 387  
 Solve\_3X3\_System(), 389  
 Sort\_RENDERLIST4DV2(), 778  
 SPHERICAL3D\_To\_POINT3D(), 344  
 SPHERICAL3D\_To\_RectXYZ(), 345  
 Start\_Clock(), 172  
 Transform\_OBJECT4DV1(), 542  
 Transform\_OBJECT4DV2(), 773  
 Transform\_RENDERLIST4DV1(), 539  
 Translate\_OBJECT4DV2(), 772  
 Translate\_Polygon2D(), 158  
 Unload\_Bitmap\_File(), 165  
 VECTOR\*D\_Add(), 346  
 VECTOR\*D\_Build(), 351  
 VECTOR\*D\_CosTh(), 352  
 VECTOR\*D\_Cross(), 349  
 VECTOR\*D\_Dot(), 349  
 VECTOR\*D\_Length(), 350  
 VECTOR\*D\_Length\_Fast(), 350  
 VECTOR\*D\_Normalize(), 351  
 VECTOR\*D\_Print(), 353  
 VECTOR\*D\_Scale(), 348  
 VECTOR\*D\_Sub(), 347  
 VECTOR\*D\_Theta\_To\_QUAT(), 374  
 VLine\*(), 162  
 Wait\_Clock(), 173  
 WinMain(), 73  
 World\_To\_Camera\_OBJECT4DV2(), 774  
 World\_To\_Camera\_RENDERLIST4DV1(), 573  
 World\_To\_Camera\_RENDERLIST4DV2(), 774  
 Write\_Error(), 165  
 Функция обратного вызова, 89

## X

Хайнс, Эрик, 488

## Ц

Цвет, 633  
   16-битовый режим, 126  
   8-битовый режим, 126  
   Модуляция, 634  
   Сложение, 633  
 Цилиндрические координаты, 231; 322

## Ш

Шаг памяти, 122  
 Шнайдер, Филипп, 1356

## Э

Эберли, Дэвид, 1356  
 Эдельштейн, Поль, 512  
 Экранные координаты, 458; 590

*Научно-популярное издание*

**Андре Ламот**

**Программирование трехмерных игр для Windows.  
Советы профессионала по трехмерной графике  
и растеризации**

Литературный редактор *С. Г. Татаренко*

Верстка *О. В. Линник*

Художественный редактор *С. А. Чернокозинский*

Корректор *Л. А. Гордиенко*

Издательский дом "Вильямс".  
101509, Москва, ул. Лесная, д. 43, стр. 1.  
Изд. лиц. ЛР № 090230 от 23.06.99  
Госкомитета РФ по печати.

Подписано в печать 30.04.2004. Формат 70X100/16.

Гарнитура «NewtonC». Печать офсетная.

Усл. печ. л. 78,5. Уч.-изд. л. 86,23.

Тираж 3000 экз. Заказ № 2506.

Отпечатано с диапозитивов в ФГУП "Печатный двор"

Министерства РФ по делам печати,

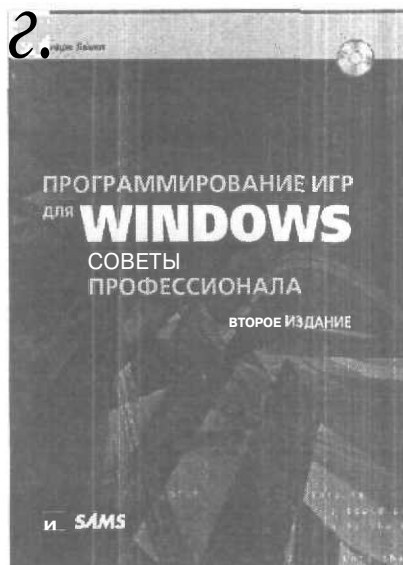
телерадиовещания и средств массовых коммуникаций.

197110, Санкт-Петербург, Чкаловский пр., 15.

# ПРОГРАММИРОВАНИЕ ИГР ДЛЯ WINDOWS

СОВЕТЫ ПРОФЕССИОНАЛА,  
2-е ИЗДАНИЕ

Андре Ламот



[www.williamspublishing.com](http://www.williamspublishing.com)

Книга предназначена для читателей, интересующихся вопросами разработки игр в операционной системе Windows. В ней освещены разнообразные аспекты программирования игр — от азов программирования до серьезного рассмотрения различных компонентов **DirectX**, от простейших физических моделей до сложных вопросов искусственного интеллекта. Книга будет полезна как начинающим, так и профессиональным разработчикам игр для Windows, хотя определенные знания в области программирования (в частности, языка программирования C или C++), математики и физики существенно облегчат изучение материала.

в продаже

# ОСВОЙ САМОСТОЯТЕЛЬНО C++.

10 МИНУТ НА УРОК

2-е ИЗДАНИЕ

**Джесс Либерти**



[www.williatnspublishing.com](http://www.williatnspublishing.com)

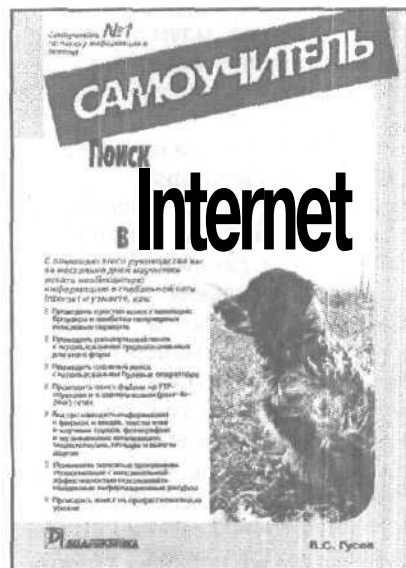
в продаже

В данной книге, задуманной как учебник для начинающих и как пособие для тех, кто уже знаком с языком C или C++, рассмотрены все важные средства языка C++: операторы и операции, обработка ошибок и исключений, управляющие конструкции, данные, управление памятью, раздельная компиляция, указатели, ссылки, ввод-вывод, классы, перегрузка функций и операторов, полиморфизм классов, объявление и определение шаблонов, Стандартная библиотека C++. Особое внимание в книге уделяется технологии программирования на языке C++. Подробно рассмотрены все этапы разработки и сопровождения программ. Теоретические положения демонстрируются на примере построения программы калькулятора.

Книга написана доступным, простым языком. Она будет полезна не только начинающим, но и тем, кто уже принимал участие в разработке больших программных проектов.

# ПОИСК В INTERNET, САМОУЧИТЕЛЬ

Гусев В. С



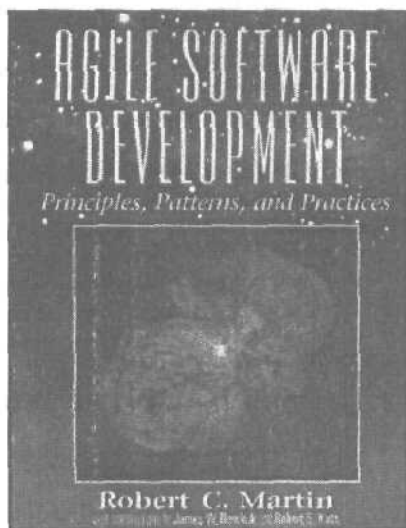
[www.dialektika.com](http://www.dialektika.com)

Самоучитель предназначен для тех, кто уже получил элементарные навыки работы в Internet и понимает: в Сети имеется огромное количество чрезвычайно полезной информации, но найти ее не так просто. В книге даны подробные рекомендации по проведению поиска разнообразных данных с помощью наиболее популярных поисковых машин, порталов, каталогов и т.д. Приведены подробные инструкции по выполнению сложных запросов на поиск и многочисленные примеры, благодаря которым даже неискушенный пользователь сможет быстро находить в Internet необходимую ему информацию.

Плановая дата выхода  
4 кв. 2003 г.

# БЫСТРАЯ РАЗРАБОТКА ПРОГРАММ: ПРИНЦИПЫ, ПРИМЕРЫ, ПРАКТИКА

**Роберт С Мартин**



[www.williamspublishing.com](http://www.williamspublishing.com)

**Плановая дата выхода  
1 кв. 2004 г.**

Имя Роберта Мартина, одного из "отцов" методик, заложенных в основу быстрого и экстремального программирования, не нуждается в представлении. Вниманию читателей предлагается его новая книга, написанная в соавторстве с Джеймсом Ньюкирком и Робертом Коссом. На сегодняшний день уже появились десятки публикаций, в которых рассматриваются современные методики разработки ПО, взятые на "вооружение" программистами. Но именно в этой книге особенно удачно продемонстрировано то, что методы экстремального программирования носят универсальный характер и не зависят от выбранной платформы и средств разработки программ. Изложение начинается с обзора основных понятий экстремального программирования и завершается готовыми программами, применяемыми на практике. В каждой главе приведены примеры кода на языках программирования Java и C++. Книга будет полезной руководителем групп программистов, нацеленных на быструю разработку коммерческих программных проектов, характеризующихся высоким уровнем качества и минимальной себестоимостью.

# БЫСТРАЯ И КАЧЕСТВЕННАЯ РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Энди Кармайкл,  
Дэн Хейвуд



[www.williamspublishing.com](http://www.williamspublishing.com)

в продаже

*Together* — это отлично интегрированный набор инструментов, позволяющий эффективно и быстро создавать программное обеспечение с высоким уровнем качества.

Книга *Быстрая и качественная разработка программного обеспечения* является полноценным руководством по наиболее эффективному использованию преимуществ *Together*. Ведущими экспертами в этой области описываются приемы, которые необходимо использовать для успешного прохождения всех этапов процесса разработки: от планирования и описания требований до разработки, отладки, передачи и управления.

Независимо от используемых технологий, аналитики, архитекторы, разработчики и менеджеры с помощью этой книги и средств *Together* смогут ускорить выполнение ближайшего проекта по разработке программного обеспечения.

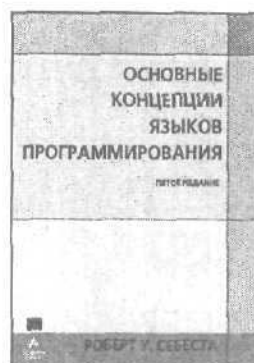
# Базовые знания программиста



ISBN 5-8459-0498-6



ISBN 5-8459-0080-8



ISBN 5-8459-0192-8



ISBN 5-8459-0360-2



ISBN 5-8459-0261-4



ISBN 5-8459-0189-8



ISBN 5-8459-0330-0



ISBN 5-8459-0579-6



ISBN 5-8459-0179-0

... и много других книг Вы найдете на наших сайтах



[www.dialektika.com](http://www.dialektika.com)



[www.williamspublishing.com](http://www.williamspublishing.com)

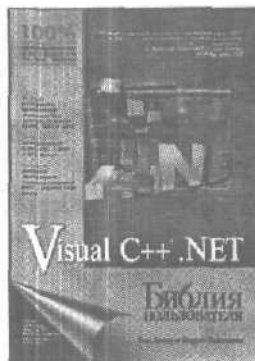


[www.ciscopress.ru](http://www.ciscopress.ru)

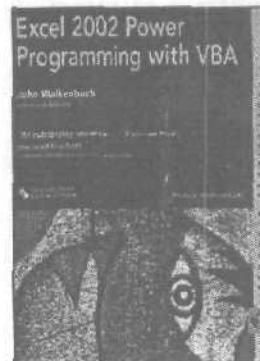
# Инструменты программиста



ISBN 5-8459-0337-8



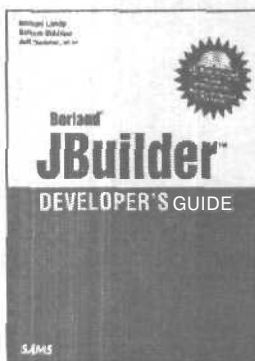
ISBN 5-8459-0462-5



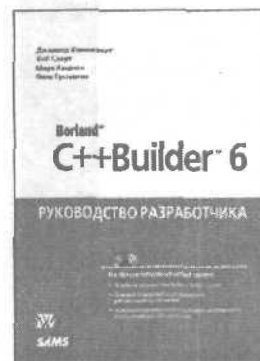
ISBN 5-8459-0541-9



ISBN 5-8459-0516-8



ISBN 5-8459-0540-0



ISBN 5-8459-0499-4



[www.dialektika.com](http://www.dialektika.com)



[www.williamspublishing.com](http://www.williamspublishing.com)



[www.ciscopress.ru](http://www.ciscopress.ru)

# Интересные темы для программиста



ISBN 5-8459-0276-2



ISBN 5-8459-0250-9



ISBN 5-8459-0437-4



ISBN 5-8459-0438-2



ISBN 5-8459-0527-3



ISBN 5-8459-0542-7



ISBN 5-8459-0558-3



ISBN 5-8459-0406-4



ISBN 5-8459-0578-8

... и много других книг Вы найдете на наших сайтах



[vsnrw.dialektika.com](http://vsnrw.dialektika.com)



[www.williamspublishing.com](http://www.williamspublishing.com)



[www.ciscopress.ru](http://www.ciscopress.ru)

Мир интересных книг от издательской группы  
**"ДИАЛЕКТИКА- ВИЛЬЯМС"**



ISBN 5-8459-0418-8



ISBN 5-8459-0461-7



ISBN 5-8459-0458-7



ISBN 5-8459-0358-0



ISBN 5-8459-0091-3



ISBN 5-8459-0079-4



ISBN 5-8459-0093-X



ISBN 5-8459-0092-1



ISBN 5-8459-0356-4



ISBN 5-8459-0005-0



ISBN 5-8459-0022-0



ISBN 5-8459-0434-X

... и много других книг Вы найдете на наших сайтах



[www.dialektika.com](http://www.dialektika.com)



[www.williamspublishing.com](http://www.williamspublishing.com)