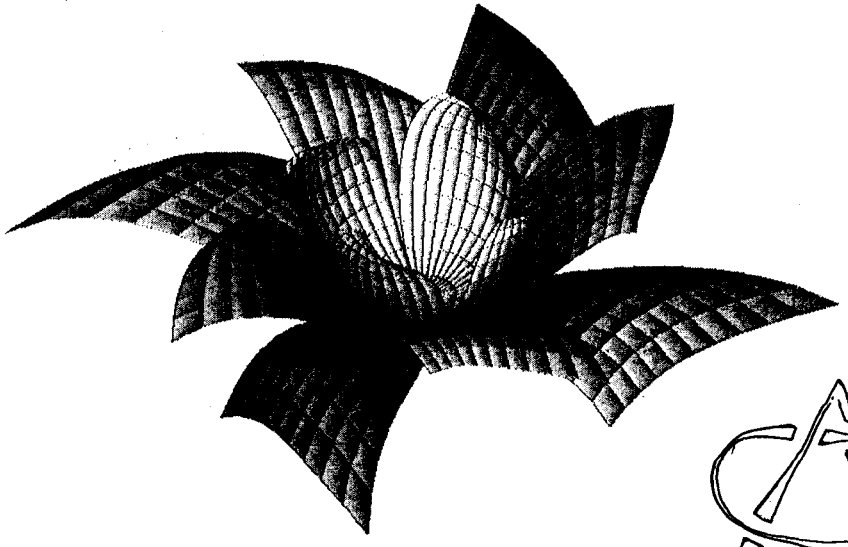
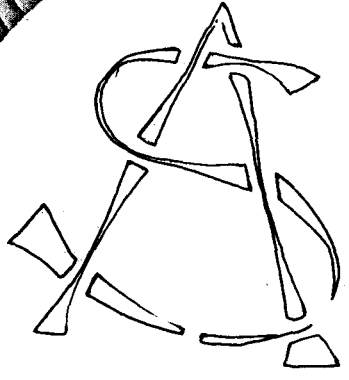
Виктор Порев

КОМПЬЮТЕРНАЯ ГРАФИКА





bhv
Санкт-Петербург

Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

В пособии рассматриваются методы и алгоритмы современной компьютерной графики. Проанализированы основные способы формирования изображений двумерных и трехмерных объектов с помощью компьютера. Освещены некоторые проблемы, возникающие перед разработчиками программного обеспечения географических информационных систем. Приведены примеры графических программ на языке C/C++ для операционной среды Windows.

Рецензент:
Доктор технических наук, профессор В. П. Широчин

Для студентов вузов, изучающих компьютерные науки, а также для разработчиков программного обеспечения

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Порев В. Н.
Компьютерная графика. — СПб.: БХВ-Петербург, 2002. — 432 с.: ил.

ISBN 5-94157-139-9

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 29.10.01.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 34,83.
Тираж 3000 экз. Заказ 1274
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99 от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12

Содержание

Предисловие 7

Введение 9

ЧАСТЬ I. ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ 15

Глава 1. Основные понятия 17

1.1. Визуализация изображений 17
1.2. Растровые изображения и их основные характеристики 18
Геометрические характеристики растра 18
Количество цветов 19
Оценка разрешающей способности растра 21
Примеры изображений для некоторых растровых устройств 22
1.3. Цвет 28
Аддитивная цветовая модель RGB 32
Цветовая модель CMY 37
Другие цветовые модели 39
Кодирование цвета. Палитра 41
Формат файлов для хранения растровых изображений 43
1.4. Методы улучшения растровых изображений 45
Устранение ступенчатого эффекта 45
Дизеринг 48
1.5. Эволюция компьютерных видеосистем 55

Глава 2. Координатный метод 63

2.1. Преобразование координат 63
Аффинные преобразования на плоскости 66
Трехмерное аффинное преобразование 69
2.2. Преобразование объектов 71
Аффинные преобразования объектов на плоскости 72
Трехмерное аффинное преобразование объектов 74

2.3. Связь преобразований объектов с преобразованиями координат 75
2.4. Проекции 79
Мировые и экранные координаты 79
Основные типы проекций 80
Аксонометрическая проекция 80
Перспективная проекция 83
Отображение в окне 90
2.5. Выводы 93

Глава 3. Базовые растровые алгоритмы 95

3.1. Алгоритмы вывода прямой линии 95
Прямое вычисление координат 97
Инкрементные алгоритмы 100
3.2. Алгоритм вывода окружности 102
3.3. Алгоритм вывода эллипса 103
3.4. Кривая Безье 105
Геометрический алгоритм для кривой Безье 107
3.5. Алгоритмы вывода фигур 107
Алгоритмы закрашивания 108
Алгоритмы заполнения, которые используют математическое описание контура 114
3.6. Стиль линии. Перо 118
Алгоритмы вывода толстой линии 119
Алгоритмы вывода пунктирной линии 121
Алгоритм вывода толстой пунктирной линии 122
3.7. Стиль заполнения. Кисть. Текстура 122
3.8. Фракталы 131

Глава 4. Методы и алгоритмы трехмерной графики 137

4.1. Модели описания поверхностей 137
Аналитическая модель 137
Векторная полигональная модель 140
Воксельная модель 145
Равномерная сетка 147
Неравномерная сетка. Изолинии 149
Преобразование моделей описания поверхности 152
4.2. Визуализация объемных изображений 159
Каркасная визуализация 161
Показ с удалением невидимых точек 161
4.3. Закрашивание поверхностей 163
Модели отражения света 163
Алгебра векторов 166
Вычисление нормалей и углов отражения 168
Метод Гуро 173
Метод Фонга 174

Преломление света ... 177
Вычисление вектора преломленного луча ... 179
Трассировка лучей ... 180

Глава 5. Примеры изображения трехмерных объектов ... 199

5.1. Шар ... 199
Каркасное изображение ... 199
Удаление невидимых точек ... 201
Многогранник с закрашиванием граней ... 201
Закрашивание граней методом Гуро ... 205
Учет расположения источника света ... 206
Градиентное закрашивание круга ... 207
Наложение текстуры на шар ... 210
Наложение текстуры на многогранник ... 214
Вариации формы шара ... 219
5.2. Цилиндр ... 222
Каркасное изображение ... 223
Удаление невидимых точек ... 223
Освещенный многогранник ... 224
Гладкая боковая поверхность ... 226
Наложение текстуры ... 229
Вариации формы цилиндра ... 232
5.3. Тор ... 234
Вариации формы тора ... 236
5.4. Общие замечания ... 237
Модель описания и способ отображения ... 237
"Квадратирование" и триангуляция ... 240

ЧАСТЬ II. ПРОГРАММИРОВАНИЕ КОМПЬЮТЕРНОЙ ГРАФИКИ ... 243

Глава 6. Разработка графических программ для Windows ... 245

6.1. Первый пример программы для Windows ... 246
6.2. Модульность программ ... 250
6.3. Использование графических функций API Windows ... 252
6.4. Контекст графического устройства ... 253
Контекст окна на экране дисплея ... 254
Контекст принтера ... 256
Контекст метафайла ... 258
Контекст памяти ... 262
Параметры контекста графического устройства ... 264

Глава 7. Графические примитивы API Windows ... 267

7.1. Отдельные пикселы ... 267
Подвижные шары ... 272

Фрактал Мандельброта 281
Трассировка лучей 284
7.2. Линии 295
Стиль линии. Перо 296
Меридианы и параллели 297
Фрактал из линий 301
7.3. Фигуры 303
Стиль заполнения. Кисть 305
Рисование поверхности 306
7.4. Шрифт TrueType 309

Глава 8. Примеры использования классов языка C++ 311

8.1. Анализ и оптимизация программы 323
8.2. И снова анализ и оптимизация программы 335

Глава 9. Пример анимации 341

9.1. Поверхность Безье 341
9.2. Градиентное закрашивание 354

Глава 10. Графическая библиотека OpenGL 359

10.1. Пример программы OpenGL 361
10.2. Координаты и матрицы 369
10.3. Пример трехмерной графики 370
10.4. Моделирование освещения 373
10.5. Стандартные объемные формы 377
10.6. Текстура 381

Приложение 389

Глоссарий 419

Список литературы 425

Предисловие

Целью написания этой книги было создать пособие по компьютерной графике для студентов, которые также изучают программирование. Занятие программированием облегчает восприятие компьютерных информационных технологий, это позволяет глубже заглянуть в мир компьютеров, получить ответы на многие вопросы типа "почему именно так". Оказалось полезным построить курс компьютерной графики с точки зрения программистов. Для лучшего восприятия этого курса желательно уже уметь, хотя бы немного, программировать на компьютерном языке C++ или C.

Материал данной книги в значительной мере соответствует курсу компьютерной графики, что читается мною в Киевском инженерно-техническом институте на протяжении последних нескольких лет. Также полезным для написания книги оказался опыт автора в качестве одного из разработчиков и программистов геоинформационной системы "ОКО" в фирме "Геобиономика".

Я понимаю, что это безнадежное дело — охватить в довольно небольшой по объему книге огромный массив знаний, накопленный в области компьютерной графики. Тем более, что уже издано достаточное количество книг на эту тему, многие из них считаю непревзойденными. Однако компьютерная графика бурно развивается, поэтому есть потребность обобщить некоторые аспекты современного состояния, написать о них как можно более простым и ясным языком, четко и понятно объяснить суть вещей. Кроме того, надеюсь, и мой собственный опыт программирования графики может кому-то быть полезным.

В любом обществе можно встретить людей, которые не удовлетворяются только использованием результатов труда других. Эта книга в первую очередь для тех, кто хочет создавать что-то собственноручно и получать удовле-

творение от процесса созидания. Компьютерная графика дает бесконечный простор для творчества. Кроме того, она возбуждает воображение человека — это один из факторов ее популярности.

Выражаю благодарность всем тем, кто, так или иначе, помогал мне в работе над книгой. Это сотрудники Киевского инженерно-технического и Киевского политехнического институтов, работники фирмы "Геобиономика". Особая благодарность жене Татьяне, без которой этой книги никогда бы не было.

Порев Виктор Николаевич

Введение

Самая важная функция компьютера — обработка информации. Особо можно выделить обработку информации, связанную с изображениями. Она разделяется на три основные направления: компьютерная графика (КГ), обработка изображений и распознавание изображений [19].

Задача **компьютерной графики** — визуализация, то есть создание изображения. Визуализация выполняется исходя из описания (модели) того, что нужно отображать. Существует много методов и алгоритмов визуализации, которые различаются между собою в зависимости от того, что и как отображать. Например, отображение того, что может быть только в воображении человека — график функций, диаграмма, схема, карта. Или наоборот, имитация трехмерной реальности — изображения сцен в компьютерных развлечениях, художественных фильмах, тренажерах, в системах архитектурного проектирования. Важными и связанными между собою факторами здесь являются: скорость изменения кадров, насыщенность сцены объектами, качество изображения, учет особенностей графического устройства.

Обработка изображений — это преобразование изображений. То есть входными данными является изображение, и результат — тоже изображение. Примером обработки изображений могут служить: повышение контраста, четкости, коррекция цветов, редукция цветов, сглаживание, уменьшение шумов и так далее. В качестве материала для обработки могут быть космические снимки, отсканированные изображения, радиолокационные, инфракрасные изображения и тому подобное. Задачей обработки изображений может быть как улучшение в зависимости от определенного критерия (реставрация, восстановление), так и специальное преобразование, кардинально изменяющее изображения. В последнем случае обработка изображений может быть промежуточным этапом для дальнейшего распознавания изображения. Например, перед распознаванием часто необходимо выделять контуры, создавать бинарное изображение, разделять по цветам. Методы обработки изо-

бражения могут существенно отличаться в зависимости от того, каким путем оно получено — синтезировано системой КГ, либо это результат оцифровки черно-белой или цветной фотографии.

Для **распознавания изображений** основная задача — получение описания изображенных объектов. Методы и алгоритмы распознавания разрабатывались, прежде всего, для обеспечения зрения роботов и для систем специального назначения. Но в последнее время компьютерные системы распознавания изображений все чаще появляются в повседневной практике многих людей — например, офисные системы распознавания текстов или программы векторизации, создание трехмерных моделей человека.

Цель распознавания может формулироваться по-разному — выделение отдельных элементов (например, букв текста на изображении документа или условных знаков на изображении карты); классификация изображения в целом (например, проверка, изображен ли определенный воздушный аппарат, или установление персоны по отпечаткам пальцев).

Методы классификации и выделения отдельных элементов могут быть взаимосвязаны. Так, классификация может быть выполнена на основе структурного анализа отдельных элементов объекта. Или для выделения отдельных элементов можно использовать методы классификации. Задача распознавания является обратной по отношению к визуализации.

До недавнего времени достаточно популярным было словосочетание ***интерактивная компьютерная графика.*** Им подчеркивалась способность компьютерной системы создавать графику и вести диалог с человеком. Раньше системы работали в пакетном режиме — способы диалога были не развиты. В настоящее время почти любую программу можно считать системой интерактивной КГ.

Исторически первыми интерактивными системами считаются ***системы автоматизированного проектирования*** (САПР), которые появились в 60-х годах [18, 19, 28]. Они представляют собой значительный этап в эволюции компьютеров и программного обеспечения. В системе интерактивной КГ пользователь воспринимает на дисплее изображение, представляющее некоторый сложный объект, и может вносить изменения в описание (модель) объекта (рис. 1). Такими изменениями могут быть как ввод и редактирование отдельных элементов, так и задание числовых значений для любых параметров, а также иные операции по вводу информации на основе восприятия изображений.

Системы типа САПР активно используются во многих областях, например в машиностроении и электронике. Одними из первых были созданы САПР для проектирования самолетов, автомобилей, системы для разработки микро-

электронных интегральных схем, архитектурные системы. Такие системы на первых порах функционировали на достаточно больших компьютерах. Потом распространилось использование быстродействующих компьютеров среднего класса с развитыми графическими возможностями — графических рабочих станций. С ростом мощностей персональных компьютеров все чаще САПР использовали на дешевых массовых компьютерах, которые сейчас имеют достаточные быстродействие и объемы памяти для решения многих задач. Это привело к широкому распространению систем САПР.

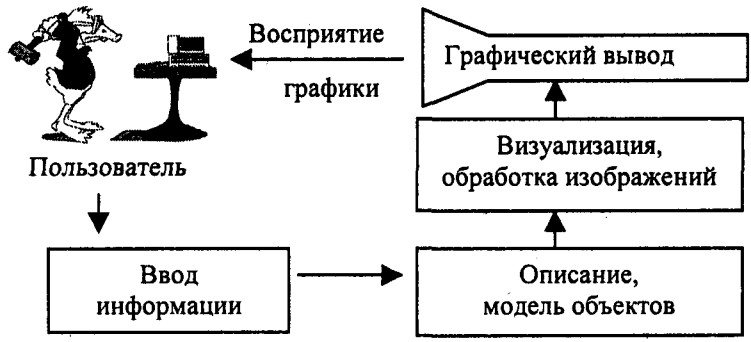


Рис. 1. Структура системы интерактивной компьютерной графики

Ныне становятся все более популярными ***геоинформационные системы*** (ГИС). Это относительно новая для массовых пользователей разновидность систем интерактивной компьютерной графики. Они аккумулируют в себе методы и алгоритмы многих наук и информационных технологий. Такие системы используют последние достижения технологий баз данных, в них заложены многие методы и алгоритмы математики, физики, геодезии, топологии, картографии, навигации и, конечно же, компьютерной графики. Системы типа ГИС зачастую требуют значительных мощностей компьютера как в плане работы с базами данных, так и для визуализации объектов, которые находятся на поверхности Земли. Причем, визуализацию необходимо делать с различной степенью детализации — как для Земли в целом, так и в границах отдельных участков (рис. 2). В настоящее время заметно стремление разработчиков ГИС повысить реалистичность изображений пространственных объектов и территорий.

Типичными для любой ГИС являются такие операции — ввод и редактирование объектов с учетом их расположения на поверхности Земли, формирование разнообразных цифровых моделей, запись в базы данных, выполнение разнообразных запросов к базам данных. Важной операцией является анализ

с учетом пространственных, топологических отношений множества объектов, расположенных на некоторой территории.

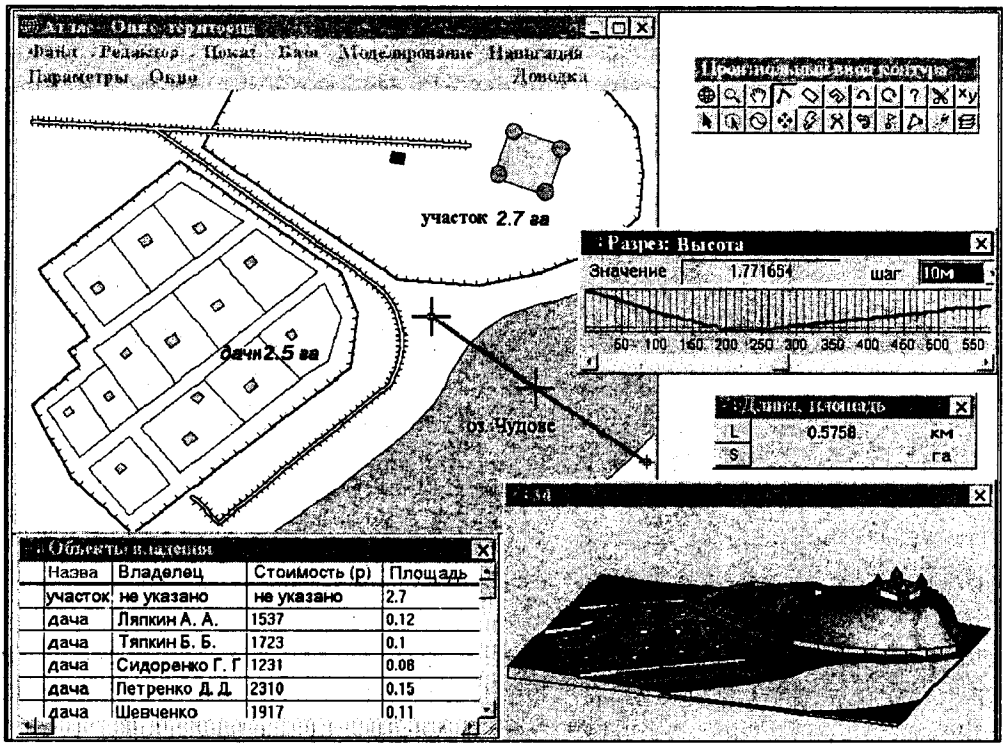


Рис. 2. Работа с геоинформационной системой (ГИС "ОКО")

Важным этапом развития систем компьютерной графики являются так называемые системы ***виртуальной реальности*** (virtual reality). Наращивание мощности компьютеров, повышение реалистичности трехмерной графики и совершенствование способов диалога с человеком позволяют создавать иллюзию вхождения человека в виртуальное пространство. Это пространство может быть моделью или существующего пространства, или выдуманного. Системы класса виртуальной реальности для диалога с компьютером обычно используют такие устройства, как шлем-дисплей, сенсоры на всем теле человека.

Образцы компьютерной графики известны уже каждому. Приобрели распространение, например, разнообразные компьютерные игры. Значительную роль в них играет анимация, реалистичность изображений, совершенство способов ввода-вывода информации. Здесь следует отметить, что во многих

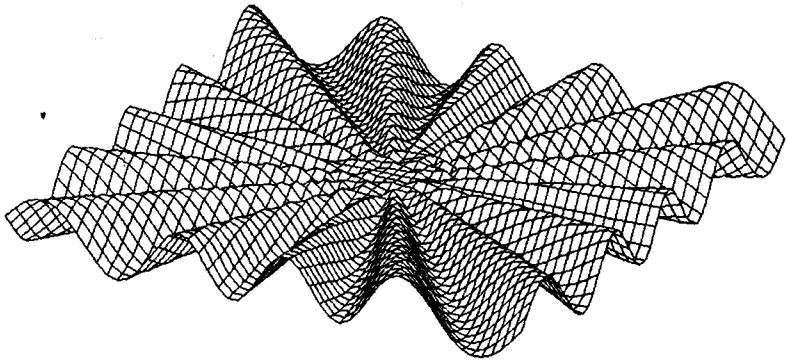
компьютерных играх реализованы идеи и методы, которые ранее были воплощены в профессиональных дорогостоящих системах, например, в тренажерах для летчиков.

Широко используется компьютерная графика в кино. Одним из первых известных фильмов был фильм "Звездные войны". Он был создан с помощью суперкомпьютера Cray. Этапы дальнейшего развития компьютерного кинематографа можно проследить по таким фильмам, как "Терминатор-2", "Вавилон 5", "Лекс" и десяткам (если уже не сотням) других. До недавнего времени технологии компьютерной графики использовались для спецэффектов, создания изображений экзотических чудовищ, имитации стихийных бедствий и других элементов, которые являлись лишь фоном для игры живых актеров. В 2001 году вышел на экраны полнометражный кинофильм "Финальная фантазия", в котором все, включая изображения людей, синтезировано компьютером — живые актеры только озвучили роли за кадром.

Важным событием в жизни общества стало появление ***глобальной сети Internet***. Сейчас происходит бурное развитие этой сети. Наращиваются мощности каналов передачи данных, усовершенствуются способы обмена и обработки информации. Сеть Internet используют все больше людей во всех странах. Это способ общения людей, обмена информацией, сближения языков, распространения идей, новое пространство для бизнеса и тому подобное. Можно относиться к Internet по-разному — например, одни считают ее важным фактором демократизации, а другие ее называют орудием чьего-то мирового господства. Вероятно, оба эти мнения справедливы, как и многие другие. Одно несомненно — создание сети Internet является выдающимся достижением человечества. Важное место в Internet занимает компьютерная графика. Все больше совершенствуются способы передачи визуальной информации, разрабатываются более совершенные графические форматы, ощутимо желание использовать трехмерную графику, анимацию, весь спектр мультимедиа.

ЧАСТЬ I

Основы компьютерной графики



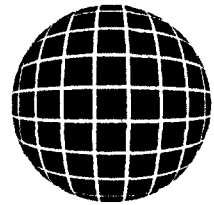
Глава 1. Основные понятия

Глава 2. Координатный метод

Глава 3. Базовые растровые алгоритмы

Глава 4. Методы и алгоритмы трехмерной графики

Глава 5. Примеры изображения трехмерных объектов



ГЛАВА 1

Основные понятия

1.1. Визуализация изображений

Наиболее известны два способа визуализации: ***растровый*** и ***векторный***. Первый способ ассоциируется с такими графическими устройствами, как дисплей, телевизор, принтер. Второй используется в векторных дисплеях, плоттерах.

Наиболее удобно, когда способ описания графического изображения соответствует способу визуализации. Иначе нужна конвертация. Например, изображение может храниться в растровом виде, а его необходимо вывести (визуализировать) на векторном устройстве. Для этого нужна предварительная **векторизация** — преобразование из растрового в векторное описание. Или наоборот, описание изображения может быть в векторном виде, а нужно визуализировать на растровом устройстве — необходима **растеризация**.

Растровая визуализация основывается на представлении изображения на экране или бумаге в виде совокупности отдельных точек (пикселов). Вместе пикселы образуют **растр**.

Векторная визуализация основывается на формировании изображения на экране или бумаге рисованием линий (векторов) — прямых или кривых. Совокупность типов линий (графических примитивов), которые используются как базовые для векторной визуализации, зависит от определенного устройства. Типичная последовательность действий при векторной визуализации для плоттера или векторного дисплея такова: переместить перо в начальную точку (*для дисплея — отклонить пучок электронов*); опустить перо (увеличить яркость луча); переместить перо в конечную точку; поднять перо (уменьшить яркость луча).

Качество векторной визуализации для векторных устройств обуславливается точностью вывода и номенклатурой базовых графических примитивов — линий, дуг, кругов, эллипсов и других.

Доминирующим сейчас является растровый способ визуализации. Это обусловлено большей распространенностью растровых дисплеев и принтеров. Недостаток растровых устройств — дискретность изображения. Недостатки векторных устройств — проблемы при сплошном заполнении фигур, меньшее количество цветов, меньшая скорость (в сравнении с растровыми устройствами).

1.2. Растровые изображения и их основные характеристики

Растр — это матрица ячеек (пикселов). Каждый пиксел может иметь свой цвет. Совокупность пикселов различного цвета образует изображение. В зависимости от расположения пикселов в пространстве различают квадратный, прямоугольный, гексагональный или иные типы растра. Для описания расположения пикселов используют разнообразные системы координат. Общим для всех таких систем является то, что координаты пикселов образуют дискретный ряд значений (необязательно целые числа). Часто используется система целых координат — номеров пикселов с (0,0) в левом верхнем уголку. Такую систему мы будем использовать и в дальнейшем, ибо она удобна для рассмотрения алгоритмов графического вывода.

Какие основные характеристики растровых изображений?

Геометрические характеристики растра

Разрешающая способность. Она характеризует расстояние между соседними пикселами (рис. 1.1). Разрешающую способность измеряют количеством пикселов на единицу длины. Наиболее популярной единицей измерения является **dpi** (dots per inch) — количество пикселов в одном дюйме длины (2.54 см). Не следует отождествлять шаг с размерами пикселов — размер пикселов может быть равен шагу, а может быть как меньше, так и больше, чем шаг.

Размер растра обычно измеряется количеством пикселов по горизонтали и вертикали. Можно сказать, что для компьютерной графики зачастую наиболее удобен растр с одинаковым шагом для обеих осей, то есть dpiX = dpiY. Это удобно для многих алгоритмов вывода графических объектов. Иначе — проблемы. Например, при рисовании окружности на экране дисплея EGA (устаревшая модель компьютерной видеосистемы, ее растр — прямоугольный, пикселы растянуты по высоте, поэтому для изображения окружности необходимо генерировать эллипс).

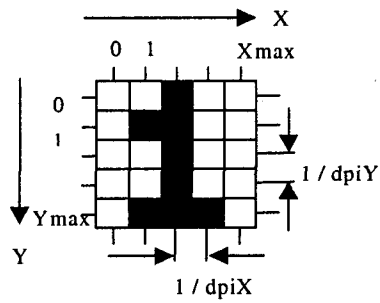


Рис. 1.1. Растр

Форма пикселов растра определяется особенностями устройства графического вывода (рис. 1.2). Например, пикселы могут иметь форму прямоугольника или квадрата, которые по размерам равны шагу растра (дисплей на жидких кристаллах); пикселы круглой формы, которые по размерам могут и не равняться шагу растра (принтеры).

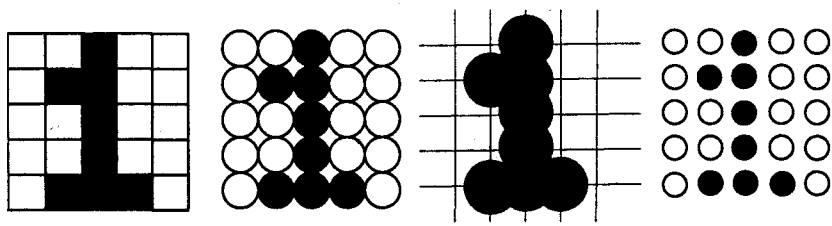


Рис. 1.2. Примеры показа одного и того же изображения на различных растрах

Количество цветов

Количество цветов (глубина цвета) — также одна из важнейших характеристик растра. Количество цветов является важной характеристикой для любого изображения, а не только растрового. Согласно психофизиологическим исследованиям глаз человека способен различать 350 000 цветов [28].

Классифицируем изображения следующим образом:

- Двухцветные (бинарные) — 1 бит на пиксел. Среди двухцветных чаще всего встречаются черно-белые изображения.
- Полутоновые — градации серого или иного цвета. Например, 256 градаций (1 байт на пиксел).
- Цветные изображения. От 2 бит на пиксел и выше. Глубина цвета 16 бит на пиксел (65 536 цветов) получила название **High Color**, 24 бит на пиксел

(16,7 млн цветов) — **True Color**. В компьютерных графических системах используют и большую глубину цвета — 32, 48 и более бит на пиксел.

В качестве примера рассмотрим растровый рисунок (рис. 1.3). Количество цветов — 256 градаций серого, разрешающая способность — примерно 100 dpi. Отметим, что в книге вы видите печатное изображение, поэтому о количестве цветов и разрешающей способности можно говорить лишь условно.

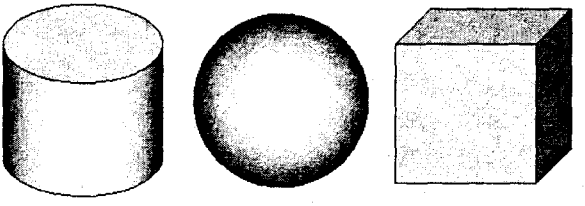


Рис. 1.3. 256 градаций серого, разрешающая способность 100 dpi

Изображения тех же объектов, но для иных параметров растра даны на рис. 1.4 и 1.5.

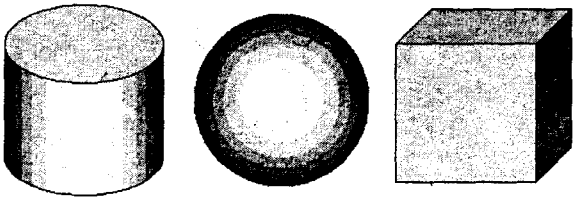


Рис. 1.4. Количество цветов (градаций серого) равно 8

Недостаточное количество цветов приводит к появлению лишних контуров на гладких поверхностях цилиндра и шара.

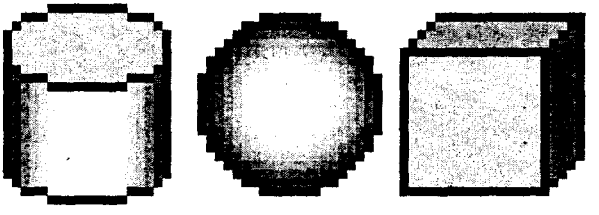


Рис. 1.5. Количество цветов здесь сохранено (256 градаций), а разрешающая способность уменьшена в 8 раз

Оценка разрешающей способности растра

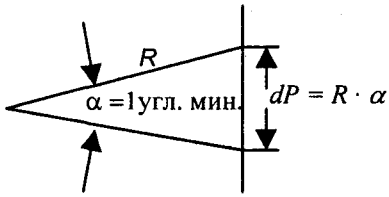


Рис. 1.6. Минимальный видимый размер

Глаз человека с нормальным зрением может различать объекты с угловым размером около одной минуты. Если расстояние до объекта равно R, то можно приблизительно оценить этот размер (dP) как длину дуги, равную $R \cdot \alpha$ (рис. 1.6). Можно предположить, что человек различает дискретность растра (шаг) также соответственно этому минимально различимому размеру. Иначе говоря, отдельные точки (пикселы), смещенные менее чем на dP, уже не воспринимаются смещенными. Тогда можно оценить разрешающую способность растра, который не воспринимается как растр, следующей величиной:

$$\mathbf{dpi} = 25.4 / \boldsymbol{dP} \text{ [мм]}.$$

Приведем несколько значений dpi для различных R (табл. 1.1).

Таблица 1.1

Расстояние *R*, мм	Размер *dP*, мм	Разрешающая способность dpi
500	0.14	181
300	0.09	282

Если считать расстояние, с которого человек обычно разглядывает бумажные документы, равным 300 мм, то можно оценить минимальную разрешающую способность, при которой уже не заметны отдельные пикселы, как приблизительно 300 dpi (примерно 0,085 мм). Лазерные черно-белые принтеры полностью удовлетворяют такому требованию.

Дисплеи обычно рекомендуется разглядывать с расстояния не ближе 0.5 м. В соответствии с приведенной выше оценкой минимальной разрешающей способности расстоянию 0,5 м соответствует около 200 dpi. В современных дисплеях минимальный размер пикселов (пятна) примерно 0,25 мм, что дает 100 dpi — это плохо, например, дисплей 15" по диагонали должен обеспечивать не 1024 на 768 пикселов, а вдвое больше. Но на современном уровне техники это пока что невозможно.

Примеры изображений для некоторых растровых устройств

Для иллюстрации работы реальных растровых устройств рассмотрим результаты вывода одной и той же картинки. Поскольку в этой книге невозможно показать цветные изображения, то в качестве тестового образца выбран черно-белый рисунок, состоящий из текста и простейшей графики. Образец изображен на рис. 1.7 в натуральную величину. Текст ("Строчка текста") набран шрифтом TrueType Times New Roman, размер 8 пунктов. Графика — векторный рисунок из линий минимально возможной толщины. Тестовый образец изготовлен и выводился на устройства с помощью редактора Word 97.

Рис. 1.7. Образец

Почему именно такой образец? Для того чтобы оценить погрешности вывода, тест следует подобрать так, чтобы устройства работали в режиме, близкому к предельно возможному. Тогда и можно оценить их возможности. Однако задача усложняется тем, что проверяются устройства различного класса. Оказалось, что некоторые устройства почти не в состоянии удовлетворительно отобразить даже такой простой образец, а некоторые устройства продемонстрировали значительный запас точности — для них нужны другие тесты.

После вывода образца на графическом устройстве соответствующее растровое изображение оцифровывалось сканером с оптическим разрешением 600×600 dpi (2400×2400 в режиме интерполяции). Отсканированные изображения в увеличенном масштабе приведены ниже. Безусловно, погрешность сканера существенна для изображений, полученных устройствами, обладающих соизмеримым, а также более высоким разрешением. Однако полученные здесь результаты не следует рассматривать как точные измерения. Здесь ставилась иная цель — проиллюстрировать геометрические свойства растров (расположение, форму и размеры отдельных пикселов) для устройств различного типа, показать наиболее характерные особенности отображения.

Для сравнения были выбраны графические устройства, которые можно встретить практически в любом современном офисе, — это дисплеи и принтеры.

Торговые марки устройств не приводятся. Приведенные образцы не следует рассматривать как тестирование или рекламу.

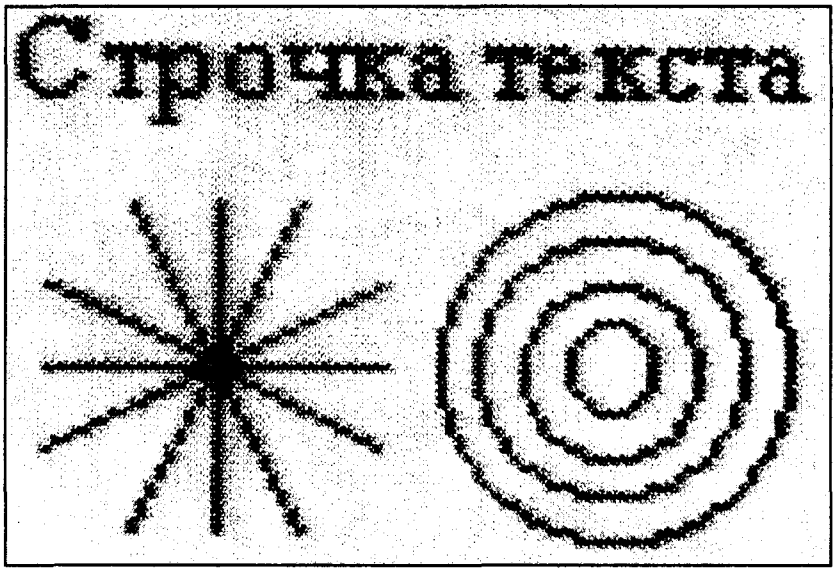


Рис. 1.8. Монитор на электронно-лучевой трубке.
Экран 15", пятно 0.28 мм, видеорежим 800 на 600, 24 бит цвет

Изображение, полученное на экране, было сфотографировано и оцифровано. Картинка имеет множество градаций серого цвета, которые здесь, к сожалению, не могут быть точно воспроизведены. Для печати иллюстраций на бумаге (в том числе и для этой книги) используется дизеринг — имитация оттенков серого цвета многими близко расположенными черными точками. Поэтому при просмотре с помощью лупы данная картинка "рассыпается" на отдельные мелкие точки. На рис. 1.9 показано увеличенное изображение фрагмента рис. 1.8. Здесь уже отчетливо видна шестиугольная структура растра, характерная для цветного кинескопа.

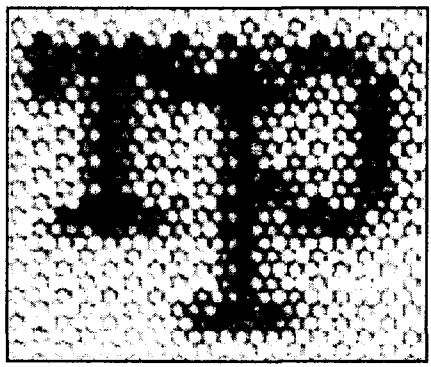


Рис. 1.9. Растр электронно-лучевой трубки

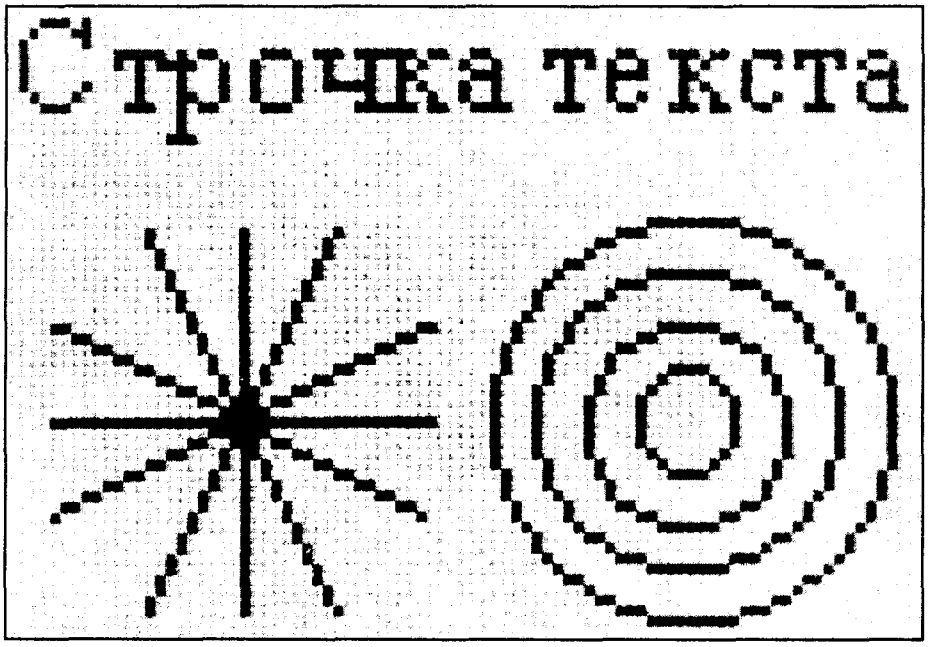


Рис. 1.10. Монитор на жидких кристаллах.
Экран ноутбука 14", видеорежим 1024 на 768, 24 бит цвет

Растровый характер изображения монитора на жидких кристаллах (рис. 1.10) выражен значительно четче, нежели монитора на электронно-лучевой трубке. Четкость отдельных пикселов обуславливает заметный лестничный эффект наклонных линий (рис. 1.11).

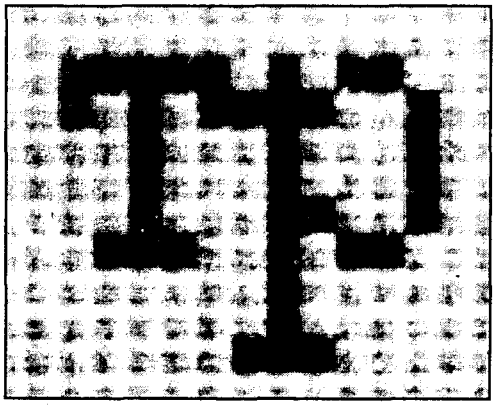


Рис. 1.11. Растр монитора на жидких кристаллах

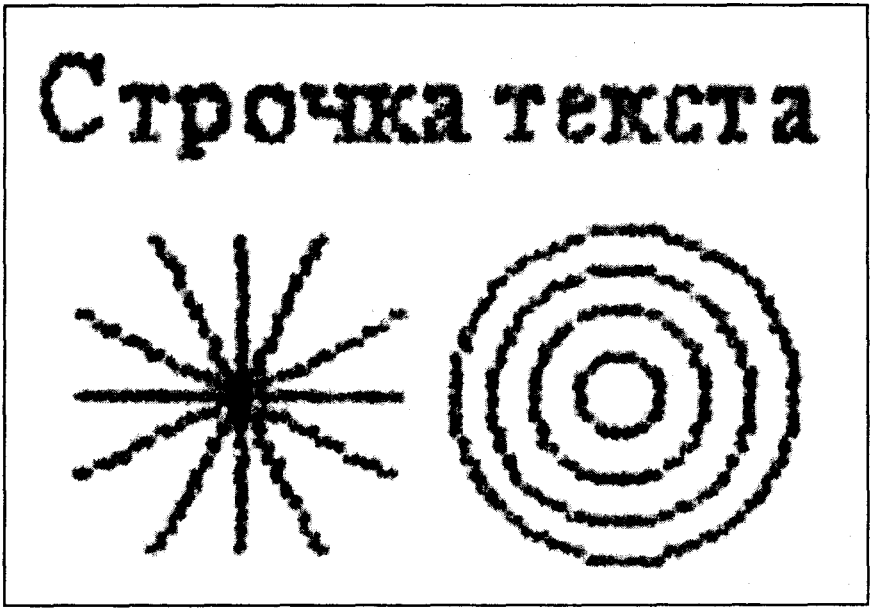


Рис. 1.12. Матричный 9-игольчатый принтер 120 на 144 dpi

Качество печати для матричных принтеров определяется погрешностями механики и износом красящей ленты. Здесь красящая лента выработала свой ресурс наполовину, поэтому изображение получилось как бы "в градациях серого цвета". Кроме того, полутоновый характер изображение имеет и из-за того, что чернота уменьшается на краях впадин оттиска игл (рис. 1.13). Вообще говоря, матричные принтеры могут печатать и намного лучше. Даже испытуемый принтер может печатать с разрешением 240×216 dpi. Однако драйвер для Windows позволяет установить только 240×144 dpi, а качество практически не улучшается по сравнению с 120×144 dpi (вероятно из-за износа механики).

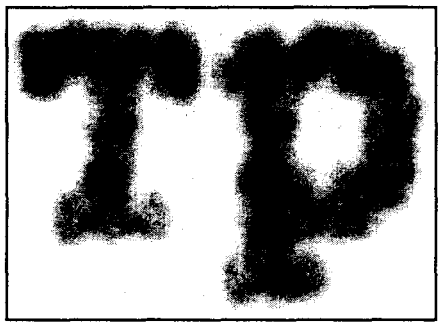


Рис. 1.13. Увеличенный фрагмент

Рис. 1.14. Лазерный черно-белый принтер, 600 dpi

Лазерные принтеры, как правило, безупречно отрабатывают свое паспортное разрешение. Немаловажным является то, что качество печати стабильно и практически не зависит от качества бумаги. Принтеры данного типа вне конкуренции (по крайней мере, в настоящее время) по быстродействию и качеству черно-белой печати среди других типов принтеров. Более дорогие модели лазерных принтеров обладают в несколько раз большим разрешением, при этом качество печати, как правило, возрастает соответственно паспортному разрешению. Оптического разрешения сканера в 600 dpi (2400 dpi интерполяция) уже не достаточно, чтобы точно отобразить фрагмент растра в мельчайших деталях (рис. 1.15).

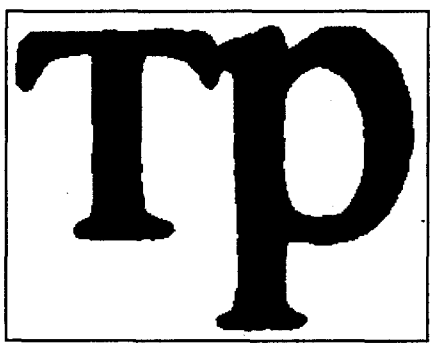


Рис. 1.15. Фрагмент растра

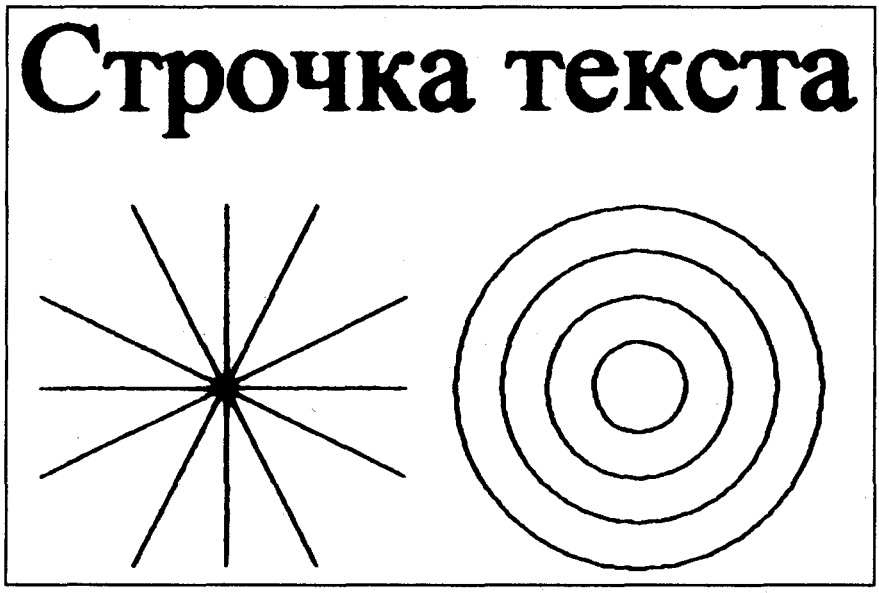


Рис. 1.16. Струйный цветной фотопринтер, черно-белый режим, 1440 dpi, печать на специальной принтерной фотобумаге

Струйные принтеры достаточно редко соответствуют заявляемой паспортной разрешающей способности. Данная модель, возможно, исключение из общего правила. В черно-белом режиме здесь фактически продемонстрирована точность печати на уровне 600 dpi лазерного принтера (рис. 1.17). Многие другие струйные принтеры с рекламируемым разрешением более тысячи dpi работают еще хуже. И это при печати на специальной бумаге.

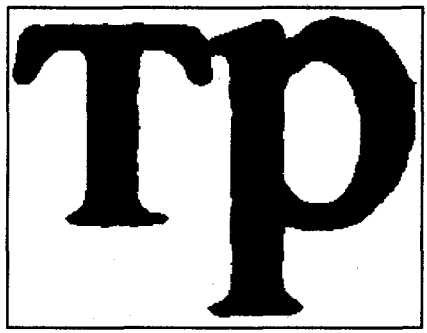


Рис. 1.17. Фрагмент изображения

Достоинством струйных принтеров является то, что это относительно недорогое устройство для цветной печати. С приемлемым качеством для цветной

фотографии работают струйные фотопринтеры. Технология струйной печати также используется и в достаточно популярных крупноформатных (A3-A1) цветных растровых принтерах.

1.3. Цвет

Для изучения способов представления цвета в компьютерных системах вначале рассмотрим некоторые общие аспекты.

Цвет — это один из факторов нашего восприятия светового излучения. Светом и цветом исследователи интересовались давно. Одним из первых выдающихся достижений в этой области являются опыты *Исаака Ньютона* в 1666 г. по разложению белого света на составляющие. Ранее считалось, что белый свет является простейшим. Ньютон опроверг это. Суть опытов Ньютона такова. Белый луч света (использовался солнечный свет) направлялся на стеклянную треугольную призму. Пройдя сквозь призму, луч преломлялся и, будучи направленный на экран, давал в результате цветную полосу — спектр. В спектре присутствовали все цвета радуги, плавно переходящие друг в друга. Эти цвета уже не раскладывались на составляющие. Ньютон разбил весь спектр на семь участков, соответствующих ярко выраженным различным цветам. Он считал эти семь цветов основными — красный, оранжевый, желтый, зеленый, голубой, синий и фиолетовый. Почему именно семь? Некоторые объясняют это убежденностью Ньютона в мистических свойствах семерки [10].

Вторая часть опытов Ньютона такова. Лучи, прошедшие сквозь призму, направлялись на вторую призму, с помощью которой удалось вновь получить белый свет. Таким образом, было доказано, что белый цвет является смесью множества различных цветов. Семь основных цветов Ньютон расположил по кругу (рис. 1.18).

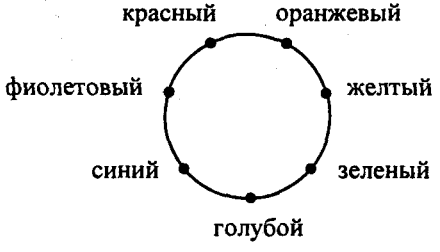


Рис. 1.18. Цветовой круг Ньютона

Ньютон предположил, что некоторый цвет образуется путем смешивания основных цветов, взятых в определенной пропорции. Если в точках на границе цветового круга, соответствующих основным цветам, расположить грузы, пропорциональные количеству каждого цвета в смеси, то суммарный цвет будет соответствовать точке центра тяжести. Белый цвет соответствует центру цветового круга [15].

Последующие исследования цвета выполняли *Томас Юнг*, *Джемс Максвелл* и другие ученые. Исследования человеческого цветовосприятия являлись достаточно важной задачей, но основные усилия были направлены на изучение объективных свойств света. В настоящее время физики полагают, что свет имеет двойственный характер. С одной стороны, свет представляется в виде потока частиц (еще Ньютон выдвинул так называемую корпускулярную теорию). С другой стороны, свету присущи волновые свойства. С помощью волновой теории, выдвинутой *Христианом Гюйгенсом* в 1678 году, были объяснены многие свойства света, в частности законы отражения и преломления [30].

Рассмотрим цвет с позиций волновых свойств. Одной из волновых характеристик света является длина волны — расстояние, которое проходит волна в течение одного периода колебания. *Монохроматическим* называется излучение, спектр которого состоит из единственной линии, соответствующей единственной длине волны. Радуга, полученная Ньютоном, состоит из бесчисленного множества монохроматических излучений (равно как и радуга, наблюдаемая нами после дождя). Достаточно качественным источником монохроматического излучения является лазер — именно поэтому его луч легко сфокусировать. Цвет монохроматического излучения определяется длиной волны. Диапазон длин волн для видимого света простирается от 380—400 нм (фиолетовый) до 700—780 нм (красный). В указанном диапазоне чувствительность человеческого зрения непостоянна. Наибольшая чувствительность наблюдается для длин волн, соответствующих зеленому цвету (рис. 1.19).

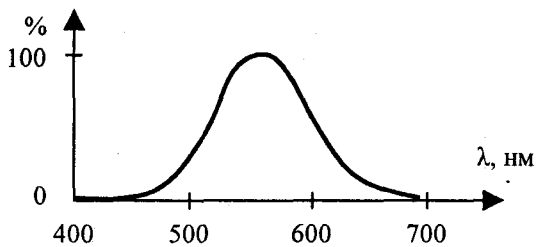


Рис. 1.19. Зависимость чувствительности человеческого зрения от длины волны светового излучения

Как показал Ньютон, белый цвет можно представить смесью всех цветов радуги. Иными словами, спектр белого является непрерывным и равномерным — в нем присутствуют излучения всех длин волн видимого диапазона.

Для характеристики цвета используются следующие атрибуты:

- ***Цветовой тон.*** Можно определить преобладающей длиной волны в спектре излучения. Цветовой тон позволяет отличать один цвет от другого — например, зеленый от красного, желтого и других.
- ***Яркость.*** Определяется энергией, интенсивностью светового излучения. Выражает количество воспринимаемого света.
- ***Насыщенность*** или чистота тона. Выражается долей присутствия белого цвета. В идеально чистом цвете примесь белого отсутствует. Если, например, к чистому красному цвету добавить в определенной пропорции белый цвет (у художников это называется разбелом), то получится светлый бледно-красный цвет.

Указанные три атрибута позволяют описать все цвета и оттенки. То, что атрибутов именно три, является одним из проявлений трехмерных свойств цвета. Как мы увидим далее, имеются и другие трехмерные системы описания цвета.

Мы попытались объяснить цвет с помощью длин волн и спектра. Как оказывается, это неполное представление о цвете, а вообще говоря, оно неправильно. Во-первых, глаз человека — это не спектроскоп. Зрительная система человека, скорее всего, регистрирует не длину волны и спектр, а формирует ощущения иным способом. Во-вторых, без учета особенностей человеческого восприятия невозможно объяснить смешение цветов. Например, белый цвет действительно можно представить равномерным спектром смеси бесконечного множества монохроматических цветов. Однако тот же белый цвет можно создать смесью всего двух специально подобранных монохроматических цветов (такие цвета называются взаимно дополнительными). Во всяком случае, человек воспринимает эту смесь как белый цвет. А можно получить белый цвет, смешав три или более монохроматических излучений. Излучения, различные по спектру, но дающие один и тот же цвет, называются *метамерными* [15].

Необходимо также уточнить, что понимается под цветовым тоном. Рассмотрим два примера спектра (рис. 1.20).

Анализ спектра, изображенного на рис. 1.20 (*а*), позволяет утверждать, что излучение имеет светло-зеленый цвет, поскольку четко выделяется одна спектральная линия на фоне равномерного спектра белого. А какой цвет (цветовой тон) соответствует спектру варианта (*б*)? Здесь нельзя выделить в

спектре преобладающую составляющую, поскольку присутствуют красная и зеленая линии одинаковой интенсивности. По законам смешения цветов это может дать оттенок желтого цвета, однако в спектре нет соответствующей линии монохроматического желтого. Поэтому под цветовым тоном следует понимать цвет монохроматического излучения, соответствующего суммарному цвету смеси. Впрочем, как именно "соответствующего" — это также требует уточнения.

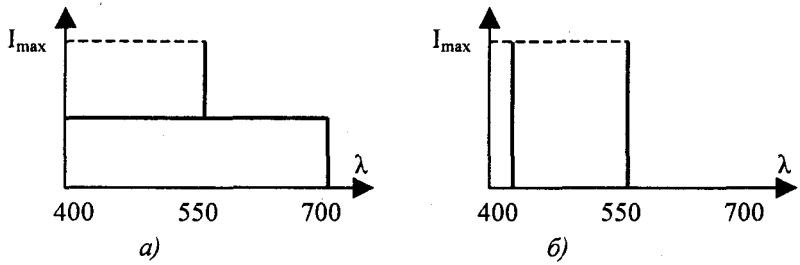


Рис. 1.20. Два спектра: *а* — имеется явное преобладание одной составляющей, *б* — две составляющие с одинаковой интенсивностью

Наука, которая изучает цвет и его измерения, называется ***колориметрией***. Она описывает общие закономерности цветового восприятия света человеком [1, 10, 15].

Одними из основных законов колориметрии являются законы смешивания цветов. Эти законы в наиболее полном виде были сформулированы в 1853 году немецким математиком *Германом Грассманом*:

1. *Цвет трехмерен — для его описания необходимы три компоненты. Любые четыре цвета находятся в линейной зависимости, хотя существует неограниченное число линейно независимых совокупностей из трех цветов.*

Иными словами, для любого заданного цвета (*Ц*) можно записать такое цветовое уравнение, выражающее линейную зависимость цветов:

$$Ц = к_1 Ц_1 + к_2 Ц_2 + к_3 Ц_3,$$

где $Ц_1$, $Ц_2$, $Ц_3$ — некоторые базисные, линейно независимые цвета, коэффициенты $к_1$, $к_2$ и $к_3$ указывают количество соответствующего смешиваемого цвета. Линейная независимость цветов $Ц_1$, $Ц_2$, $Ц_3$ означает, что ни один из них не может быть выражен взвешенной суммой (линейной комбинацией) двух других.

Первый закон можно трактовать и в более широком смысле, а именно, в смысле трехмерности цвета. Необязательно для описания цвета применять

смесь других цветов, можно использовать и другие величины — но их обязательно должно быть три.

2. *Если в смеси трех цветовых компонент одна меняется непрерывно, в то время, как две другие остаются постоянными, цвет смеси также изменяется непрерывно.*
3. *Цвет смеси зависит только от цветов смешиваемых компонент и не зависит от их спектральных составов.*

Смысл третьего закона становится более понятным, если учесть, что один и тот же цвет (в том числе и цвет смешиваемых компонент) может быть получен различными способами. Например, смешиваемая компонента может быть получена, в свою очередь, смешиванием других компонент.

Аддитивная цветовая модель RGB

Эта модель используется для описания цветов, которые получаются с помощью устройств, основанных на принципе излучения. В качестве основных цветов выбран красный (Red), зеленый (Green) и синий (Blue). Иные цвета и оттенки получаются смешиванием определенного количества указанных основных цветов.

Рис. 1.21. Основные цвета RGB и их смешивание

Вкратце история системы RGB такова. *Томас Юнг* (1773—1829) взял три фонаря и приспособил к ним красный, зеленый и синий светофильтры. Так были получены источники света соответствующих цветов. Направив на белый экран свет этих трех источников, ученый получил такое изображение (рис. 1.21). На экране свет от источников давал цветные круги. В местах пересечения кругов наблюдалось смешивание цветов. Желтый цвет получался смешиванием красного и зеленого, голубой — смесь зеленого и синего, пурпурный — синего и красного, а белый цвет образовывался смешением всех

трех основных цветов. Некоторое время спустя, *Джемс Максвелл* (1831—1879) изготовил первый колориметр, с помощью которого человек мог зрительно сравнивать монохроматический цвет и цвет смешивания в заданной пропорции компонент RGB. Регулируя яркость каждой из смешиваемых компонент, можно добиться уравнивания цветов смеси и монохроматического излучения. Это описывается следующим образом:

$$Ц = r\,R + g\,G + b\,B,$$

где r, g и b — количество соответствующих основных цветов.

Соотношение коэффициентов r, g и b Максвелл наглядно показал с помощью треугольника, впоследствии названного его именем [1]. Треугольник Максвелла является равносторонним, в его вершинах располагаются основные цвета — R, G и B (рис. 1.22). Из заданной точки проводятся линии, перпендикулярные сторонам треугольника. Длина каждой линии и показывает соответствующую величину коэффициента r, g или b. Одинаковые значения $r=g=b$ имеют место в центре треугольника и соответствуют белому цвету. Следует также отметить, что некоторый цвет может изображаться как внутренней точкой такого треугольника, так и точкой, лежащей за его пределами. В последнем случае это соответствует отрицательному значению соответствующего цветового коэффициента. Сумма коэффициентов равна высоте треугольника, а при высоте, равной единице, — $r+g+b = 1$.

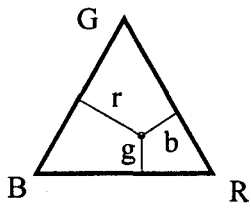


Рис. 1.22. Треугольник Максвелла

В качестве основных цветов Максвелл использовал излучения с такими длинами волн — 630, 528 и 457 нм.

К настоящему времени система RGB является официальным стандартом. Решением Международной Комиссии по Освещению — МКО (CIE — Commision International de l'Eclairage) в 1931 году были стандартизованы основные цвета, которые было рекомендовано использовать в качестве R, G и B. Это монохроматические цвета светового излучения с длинами волн соответственно:

R — 700 нм; G — 546.1 нм; B — 435.8 нм.

Красный цвет получается с помощью лампы накаливания с фильтром. Для получения чистых зеленого и синего цветов используется ртутная лампа. Также стандартизировано значение светового потока для каждого основного цвета [1].

Еще одним важным параметром для системы RGB является цвет, получаемый смешением трех компонент в равных количествах. Это белый цвет. Оказывается, что для того, чтобы смешиванием компонент R, G и B получить белый цвет, яркости соответствующих источников не должны быть равны друг другу, а находиться в пропорции

$$L_R : L_G : L_B = 1 : 4.5907 : 0.0601.$$

Если расчеты цвета производятся для источников излучения с одинаковой яркостью, то указанное соотношение яркостей можно учесть соответствующими масштабными коэффициентами [15].

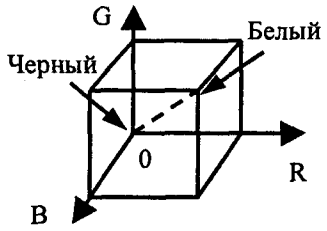


Рис. 1.23. Трехмерные координаты RGB

Теперь рассмотрим другие аспекты. Цвет, создаваемый смешиванием трех основных компонент, можно представить вектором в трехмерной системе координат R, G и B, изображенной на рис. 1.23. Черному цвету соответствует центр координат — точка (0, 0, 0). Белый цвет выражается максимальным значением компонент. Пусть это максимальное значение вдоль каждой оси равно единице. Тогда белый цвет — это вектор (1, 1, 1). Точки, лежащие на диагонали куба от черного к белому, соответствуют равным значениям: $R_i = G_i = B_i$. Это градации серого — их можно считать белым цветом различной яркости. Вообще говоря, если все компоненты вектора (r, g, b) умножить на одинаковый коэффициент (k = 0...1), то цвет (kr, kg, kb) сохраняется, изменяется только яркость. Поэтому, для анализа цвета важно соотношение компонент. Если в цветовом уравнении

$$Ц = r\,R + g\,G + b\,B$$

разделить коэффициенты r, g и b на их сумму:

$$r' = \frac{r}{r+g+b}, \qquad g' = \frac{g}{r+g+b}, \qquad b' = \frac{b}{r+g+b},$$

то можно записать такое цветовое уравнение:

$$Ц = r'\,R + g'\,G + b'\,B.$$

Это уравнение выражает векторы цвета (r', g', b'), лежащие в единичной плоскости $r'+g'+b' = 1$. Иными словами, мы перешли от куба к треугольнику Максвелла.

В ходе колориметрических экспериментов были определены коэффициенты (r', g', b'), соответствующие чистым монохроматическим цветам. Простейший колориметр можно представить как призму из белого гипса, грани которой освещают источниками света. На левую грань направлен источник чистого монохроматического излучения, а правая грань освещается смесью трех источников RGB. Наблюдатель видит одновременно две грани, что позволяет фиксировать равенство цветов (рис. 1.24).

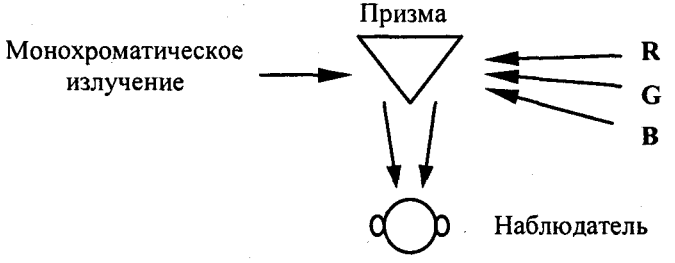


Рис. 1.24. Схема уравнивания цветов

Результаты экспериментов можно изобразить графически (рис. 1.25).

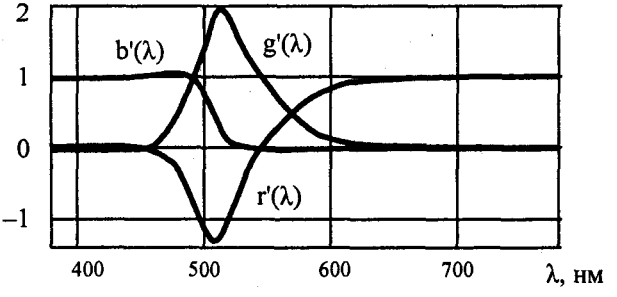


Рис. 1.25. Трехцветные коэффициенты смешивания RGB

Как видим, коэффициенты r', g' и b' могут быть и положительными, и отрицательными. Что это означает? То, что некоторые монохроматические цвета не могут быть представлены суммой компонент R, G и B. Но как отнять то, чего нет? Для уравнивания цвета пришлось добавить к монохроматическому излучению одну из компонент R, G или B. Например, если монохроматиче-

ское излучение для некоторого значения λ разбавлялось красным, то это можно выразить так:

$$Ц(\lambda) + r'(\lambda)\, R = g'(\lambda)\, G + b'(\lambda)\, B\,.$$

Как оказалось, ни один цвет монохроматического излучения (за исключением самих цветов R, G и B) не может быть представлен только положительными значениями коэффициентов смешивания. Это наглядно можно изобразить с помощью цветового графика, построенного на основе треугольника Максвелла (рис. 1.26). Верхняя часть кривой линии соответствует чистым монохроматическим цветам, а нижняя линия — от 380 до 780 нм — представляет так называемые *пурпурные* цвета (смесь синего и красного), которые не являются монохроматическими. Точки, лежащие внутри контура кривой, соответствуют реальным цветам, а вне этого контура — нереальным цветам. Точки внутри треугольника соответствуют неотрицательным значениям коэффициентов r', g' и b' и представляют цвета, которые можно получить смешиванием компонент RGB.

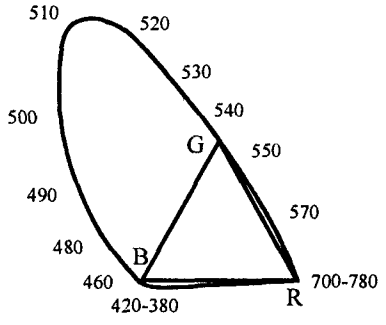


Рис. 1.26. Цветовой график RGB

Таким образом, система RGB имеет неполный цветовой охват — некоторые насыщенные цвета не могут быть представлены смесью указанных трех компонент. В первую очередь, это цвета от зеленого до синего, включая все оттенки голубого — они соответствуют левой ветви кривой цветового графика. Еще раз подчеркнем, что речь здесь идет о насыщенных цветах, поскольку, например, ненасыщенные голубые цвета смешиванием компонент RGB получить можно. Несмотря на неполный охват, система RGB широко используется в настоящее время — в первую очередь, в цветных телевизорах и дисплеях компьютеров. Отсутствие некоторых оттенков цвета не слишком заметно.

Еще одним фактором, способствующим популярности системы RGB, является ее наглядность — основные цвета находятся в трех четко различимых участках видимого спектра.

Кроме того, одной из гипотез, объясняющих цветовое зрение человека, является *трехкомпонентная теория*, которая утверждает, что в зрительной системе человека есть три типа светочувствительных элементов. Один тип элементов реагирует на зеленый, другой тип — на красный, а третий тип — на

синий цвет. Такая гипотеза высказывалась еще *Ломоносовым* [15], ее обоснованием занимались многие ученые, начиная с *Т. Юнга*. Впрочем, трехкомпонентная теория не является единственной теорией цветового зрения человека [1, 10, 15].

Цветовая модель CMY

Используется для описания цвета при получении изображений на устройствах, которые реализуют принцип поглощения (вычитания) цветов. В первую очередь она используется в устройствах, которые печатают на бумаге. Название данной модели составлено из названий основных субтрактивных цветов — голубого (Cyan), пурпурного (Magenta) и желтого (Yellow) (рис. 1.27).

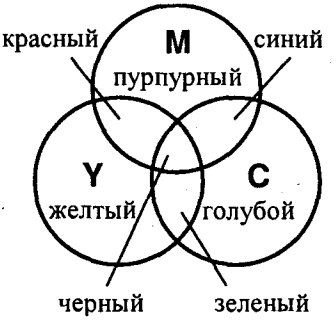


Рис. 1.27. Основные цвета системы CMY

Для того чтобы разобраться с поглощением цветов, рассмотрим рис. 1.28.

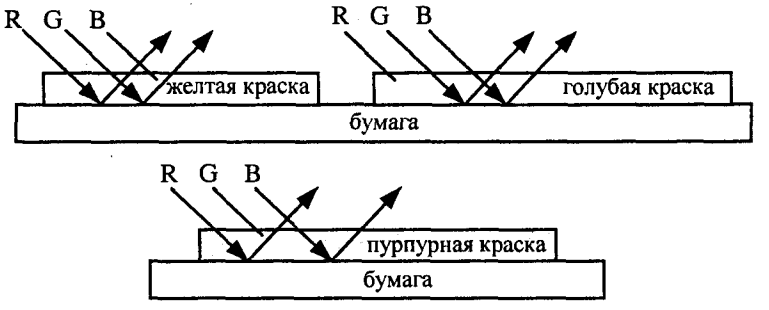


Рис. 1.28. Поглощение (вычитание) цветов

Нанесение желтой краски на белую бумагу означает, что поглощается отраженный синий цвет. Голубая краска поглощает красный цвет, пурпурная краска — зеленый.

Комбинирование красок позволяет получить цвета, которые остались — зеленый, красный, синий и черный. Черный цвет соответствует поглощению всех цветов при отражении (рис. 1.29).

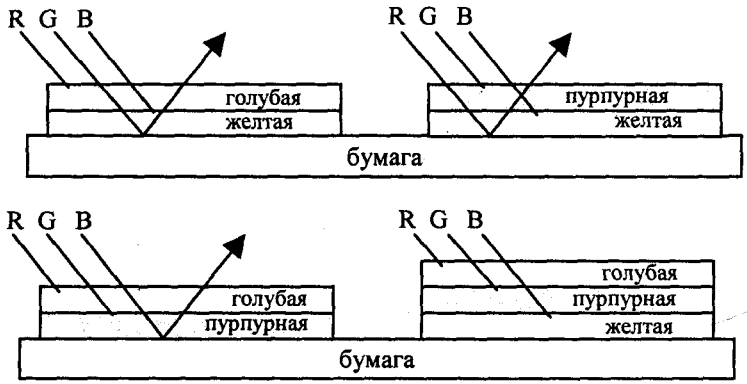


Рис. 1.29. Субтрактивность для двух и трех красок

На практике добиться черного смешиванием сложно из-за неидеальности красок, поэтому в принтерах используют еще и краску черного цвета (blacK). Тогда модель называется CMYK.

Необходимо также отметить, что не всякие краски обеспечивают указанное выше вычитание цветов CMY. Подробнее об этом в [10].

В табл. 1.2 представлены некоторые цвета в моделях RGB и CMY.

Таблица 1.2

Цвет	Модель RGB			Модель CMY		
	R	G	B	C	M	Y
Красный	1	0	0	0	1	1
Желтый	1	1	0	0	0	1
Ярко-зеленый	0	1	0	1	0	1
Голубой	0	1	1	1	0	0
Синий	0	0	1	1	1	0
Пурпурный	1	0	1	0	1	0
Черный	0	0	0	1	1	1
Белый	1	1	1	0	0	0

Соотношение для перекодирования цвета из модели CMY в RGB:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}.$$

И обратно — из модели RGB в CMY:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

Здесь считается, что компоненты кодируются числами в диапазоне от 0 до 1. Для иного диапазона чисел можно записать соответствующие соотношения.

Другие цветовые модели

Для решения проблемы отрицательных коэффициентов, которая имеет место для модели RGB, в 1931 году Международной Комиссией по Освещению (CIE) была принятая колориметрическая система **XYZ** (рис. 1.30). В системе МКО XYZ в качестве основных цветов были приняты также три цвета, однако они являются условными, нереальными [1, 15].

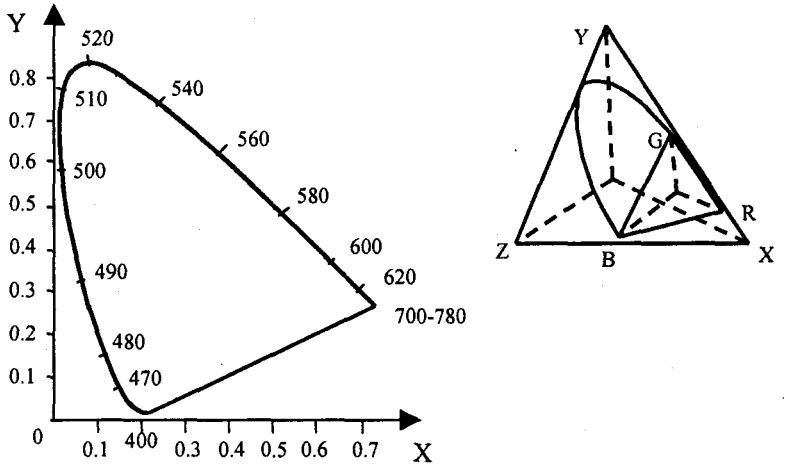


Рис. 1.30. Цветовой график для модели МКО XYZ

Рассмотренные выше цветовые модели так или иначе используют смешивание некоторых основных цветов. Теперь рассмотрим цветовую модель, которую можно отнести к иному, альтернативному типу.

В модели **HSV** цвет описывается следующими параметрами — *цветовой тон* H (hue), *насыщенность* S (saturation), *яркость, светлота* V (value). Значение H измеряется в градусах от 0 до 360, поскольку здесь цвета радуги располагаются по кругу в таком порядке — красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый (известна поговорка "каждый охотник желает знать, где сидят фазаны"). Значения S и V находятся в диапазоне (0...1).

Приведем примеры кодирования цветов для модели HSV (рис. 1.31). При S=0 (то есть на оси V) — серые тона. Значение V=0 соответствует черному цвету. Белый цвет кодируется как S=0, V=1. Цвета, расположенные по кругу напротив друг друга, то есть отличающиеся по H на 180 градусов, являются дополнительными [28]. Задание цвета с помощью параметров HSV достаточно часто используется в графических системах, причем обычно показывается развертка конуса.

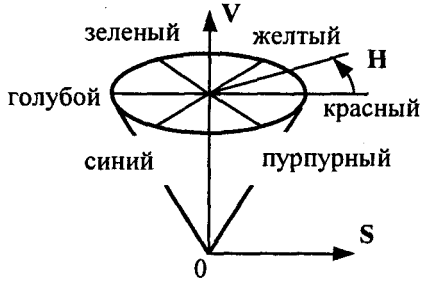


Рис. 1.31. Модель HSV

Существуют и другие цветовые модели, построенные аналогично HSV, например, модель **HLS** (Hue, Lighting, Saturation) также использует цветовой конус.

В [42] есть сведения о цветовой модели **CIE L*a*b***, которая была принятая МКО.

Все вышеперечисленные цветовые модели описывают цвет тремя параметрами. Они описывают цвет в достаточно широком диапазоне. Теперь рассмотрим цветовую модель, в которой цвет задается одним числом, но уже для ограниченного диапазона цветов (оттенков).

На практике часто используются черно-белые (серые) полутоновые изображения. Серые цвета в модели RGB описываются одинаковыми значениями компонентов, то есть $r_i = g_i = b_i$. Таким образом, для серых изображений нет необходимости использовать тройки чисел — достаточно и одного числа. Это позволяет упростить цветовую модель. Каждая градация определяется

яркостью Y. Значение Y=0 соответствует черному цвету, максимальное значение Y — белому.

В качестве примера рассмотрим преобразование цветных изображений, представленных в системе RGB, в градации серого (подобно тому, как показываются цветные фильмы на черно-белом экране телевизора). Для этого можно воспользоваться соотношением

$$Y = 0.299\,R + 0.587\,G + 0.114\,B,$$

где коэффициенты при *R*, *G* и *B* учитывают различную чувствительность зрения к соответствующим цветам и, кроме того, их сумма равна единице. Очевидно, что обратное преобразование *R=Y*, *G=Y*, *B=Y* не даст никаких других цветов, кроме градаций серого.

Еще один пример использования различных цветовых моделей. При записи цветных фотографий в графический файл формата JPEG выполняется преобразование из модели RGB в модель (Y, Cb, Cr). Это используется для дальнейшего сжатия объемов информации растрового изображения. При чтении файлов JPEG выполняется обратное преобразование в RGB.

Разнообразие моделей обусловлено различными областями их использования. Каждая из цветовых моделей была разработана для эффективного выполнения отдельных операций: ввода изображений, визуализации на экране, печати на бумаге, обработки изображений, сохранения в файлах, колориметрических расчетов и измерений. Преобразование одной модели в другую может привести к искажению цветов изображения.

Кодирование цвета. Палитра

Для того чтобы компьютер имел возможность работать с цветными изображениями, необходимо представлять цвета в виде чисел — кодировать цвет. Способ кодирования зависит от цветовой модели и формата числовых данных в компьютере.

Для модели RGB каждая из компонент может представляться числами, ограниченными некоторым диапазоном — например, дробными числами от 0 до 1 либо целыми числами от 0 до некоторого максимального значения. В настоящее время достаточно распространенным является формат True Color, в котором каждая компонента представлена в виде байта, что дает 256 градаций для каждой компоненты: *R*=0...255, *G* = 0...255, *B* = 0...255. Количество цветов составляет 256×256×256 = 16.7 млн (2^{24}).

Такой способ кодирования цветов можно назвать ***компонентным***. В компьютере коды изображений True Color представляются в виде троек байтов,

либо упаковываются в длинное целое (четырехбайтное) — 32 бита (так, например, сделано в API Windows):

$$C = 00000000\ bbbbbbbb\ gggggggg\ rrrrrrrr.$$

При работе с изображениями в системах компьютерной графики часто приходится искать компромисс между качеством изображения (требуется как можно больше цветов) и ресурсами, необходимыми для хранения и воспроизведения изображения, исчисляемыми, например, объемом памяти (надо уменьшать количество бит на пиксел).

Кроме того, некоторое изображение само по себе может использовать ограниченное количество цветов. Например, для черчения может быть достаточно двух цветов, для человеческого лица важны оттенки розового, желтого, пурпурного, красного, зеленого; а для неба — оттенки голубого и серого. В этих случаях использование полноцветного кодирования цвета является избыточным.

При ограничении количества цветов используют ***палитру***, представляющую набор цветов, важных для данного изображения. Палитру можно воспринимать как таблицу цветов. Палитра устанавливает взаимосвязь между кодом цвета и его компонентами в выбранной цветовой модели.

В качестве примера дадим стандартную палитру дисплейных 16-цветных видеорежимов EGA, VGA (табл. 1.3).

Таблица 1.3

Код цвета	R	G	B	Название цвета
0	0	0	0	Черный
1	128	0	0	Темно-красный
2	0	128	0	Зеленый
3	128	128	0	Коричнево-зеленый
4	0	0	128	Темно-синий
5	128	0	128	Темно-пурпурный
6	0	128	128	Сине-зеленый
7	128	128	128	Серый 50%
8	192	192	192	Серый 25%
9	255	0	0	Красный

Таблица 1.3 (окончание)

Код цвета	R	G	B	Название цвета
10	0	255	0	Ярко-зеленый
11	255	255	0	Желтый
12	0	0	255	Синий
13	255	0	255	Пурпурный
14	0	255	255	Голубой
15	255	255	255	Белый

Недостатком такой палитры можно считать отсутствие одного из важных цветов — оранжевого. Существуют также иные стандартные палитры, например, 256-цветная для VGA. Компьютерные видеосистемы обычно предоставляют возможность программисту установить собственную палитру.

Каждый цвет изображения, использующего палитру, кодируется индексом, который будет определять номер строки в таблице палитры. Поэтому такой способ кодирования цвета называют ***индексным***.

Формат файлов для хранения растровых изображений

К настоящему времени известно много форматов файлов для растровых изображений. Здесь мы рассмотрим один из самых популярных форматов, который обязан своей распространенностью операционной системе Windows — формат **BMP**.

Общая структура BMP-файла такова:

BITMAPFILEHEADER	14 байт
BITMAPINFOHEADER	40 байт
Палитра	Размер зависит от количества цветов
Битовый массив растрового изображения	Число байт определяется размерами растра и количеством бит на пиксел

Заголовок файла BMP называется **BITMAPFILEHEADER**. В нем помещается общее описание файла. Заголовок имеет следующие поля:

WORD **bfType** — хранит символы "BM". Это код формата.

DWORD **bfSize** — общий размер файла в байтах.

WORD **bfReserved1** — зарезервировано, пока что равно 0.

WORD **bfReserved2** — зарезервировано, пока что равно 0.

DWORD **bfOffBits** — адрес битового массива в данном файле.

Далее в файле следует еще один заголовок — **BITMAPINFOHEADER**, в котором хранится описание размеров растра и цветового формата пикселов. Здесь имеются такие поля:

DWORD **biSize** — размер заголовка, равен 40.

LONG **biWidth** — ширина растра в пикселах.

LONG **biHeight** — высота растра в пикселах.

WORD **biPlanes** — должно быть равно 1.

WORD **biBitCount** — бит на пиксел, может быть 1, 4, 8, 16, 24 или 32.

DWORD **biCompression** — равно нулю.

DWORD **biSizeImage** — размер в байтах битового массива растра.

LONG **biXPelsPerMeter** — разрешение по X в пикселах на метр.

LONG **biYPelsPerMeter** — разрешение по Y в пикселах на метр.

DWORD **biClrUsed** — если равно 0, то используется макс. число цветов.

DWORD **biClrImportant** — равно 0, если **biClrUsed** = 0.

Затем в файле помещается палитра в виде записей **RGBQUAD**. Каждая запись содержит четыре поля.

BYTE **rgbBlue** — цветовая компонента B, от 0 до 255.

BYTE **rgbGreen** — компонента G.

BYTE **rgbRed** — компонента R.

BYTE **rgbReserved** — не используется, равно 0.

Количество записей **RGBQUAD** равно количеству используемых цветов. Палитра отсутствует, если число бит на пиксел равно 24. Также палитра не нужна и для некоторых цветовых форматов 16 и 32 бит на пиксел.

Здесь приняты такие обозначения для типов полей:

BYTE — однобайтовое целое число без знака.

WORD — двухбайтовое целое число без знака.

DWORD — четырехбайтовое целое число без знака.

LONG — четырехбайтовое целое число со знаком.

После палитры (если она есть) в файле BMP записывается растр в виде битового (а точнее, байтового массива). В битовом массиве последовательно записываются байты строк растра. Количество байт в строке должно быть кратно четырем, поэтому если количество пикселов по горизонтали не соответствует такому условию, то справа в каждую строку дописывается некоторое число битов (выравнивание строк на границу двойного слова).

Сжатие изображений в файлах BMP отсутствует, однако подобный формат (Device Independent Bitmap) описания растровых изображений также используется и для файлов типа DIB, где применяются простейшие алгоритмы сжатия RLE (Run Length Encoding) [61]. Алгоритмы RLE также используются и в других популярных растровых графических форматах, например PCX.

Описание других графических форматов можно найти в литературных источниках, например [2], а также в сети Internet.

1.4. Методы улучшения растровых изображений

Рассмотрим некоторые из существующих методов улучшения качества изображений, которые основываются на субъективном восприятии разрешающей способности и количества цветов. При одних и тех же значениях технических параметров устройства графического вывода можно создать иллюзию увеличения разрешающей способности или количества цветов. Причем субъективное улучшение одной характеристики происходит за счет ухудшения другой.

Устранение ступенчатого эффекта

В растровых системах при невысокой разрешающей способности (меньше 300 dpi) существует проблема ступенчатого эффекта (**aliasing**). Этот эффект особенно заметен для наклонных линий — при большом шаге сетки растра пикселы образуют как бы ступени лестницы.

Рассмотрим это на примере отрезка прямой линии. Вообще говоря, растровое изображение объекта определяется алгоритмом закрашивания пикселов, со-

ответствующих площади изображаемого объекта. Различные алгоритмы могут дать существенно отличающиеся варианты растрового изображения одного и того же объекта. Можно сформулировать условие корректного закрашивания следующим образом — если в контур изображаемого объекта попадает больше половины площади ячейки сетки растра, то соответствующий пиксел закрашивается цветом объекта (C), иначе — пиксел сохраняет цвет фона ($C_ф$).

На рис. 1.32 показано растровое изображение толстой прямой линии, на которое для сравнения наложен идеальный контур исходной линии.

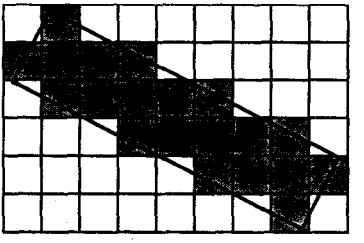


Рис. 1.32. Растровое изображение отрезка линии

Устранение ступенчатого эффекта называется по-английски **antialiasing**. Для того чтобы растровое изображение линии выглядело более гладким, можно цвет угловых пикселов "ступенек лестницы" заменить на некоторый оттенок, промежуточный между цветом объекта и цветом фона. Будем вычислять цвет пропорционально части площади ячейки растра, покрываемой идеальным контуром объекта. Если площадь всей ячейки обозначить как S, а часть площади, покрываемой контуром, — S_x, то искомый цвет равен

$$C_x = \frac{C \cdot S_x + C_ф \cdot (S - S_x)}{S}.$$

На рис. 1.33 показано сглаженное растровое изображение, построенное указанным выше методом.

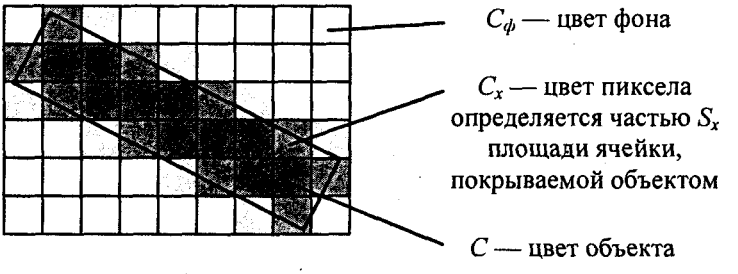


Рис. 1.33. Сглаживание

Методы получения сглаженных растровых изображений можно разделить на две группы. Первую группу составляют алгоритмы генерации сглаженных изображений отдельных простейших объектов — линий, фигур. Некоторые из таких алгоритмов описаны в [33].

Другую группу методов сглаживания составляют методы обработки уже нарисованного изображения. Для сглаживания растровых изображений часто используют алгоритмы цифровой фильтрации. Один из таких алгоритмов — локальная фильтрация. Она осуществляется путем взвешенного суммирования яркостей пикселов, расположенных в некоторой окрестности текущего обрабатываемого пиксела. Можно представить себе, что в ходе обработки по растру скользит прямоугольное окно, которое выхватывает пикселы, используемые для вычисления цвета некоторого текущего пиксела. Если окрестность симметрична, то текущий пиксел находится в центре окна (рис. 1.34).

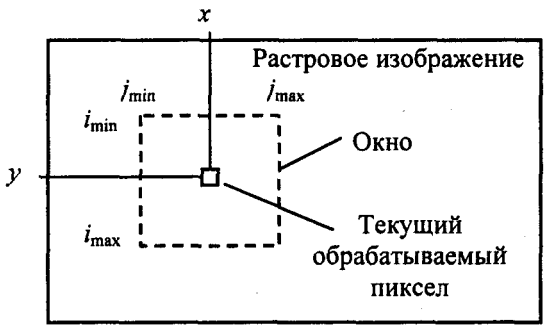


Рис. 1.34. Окно — окрестность обрабатываемого пиксела

Базовую операцию такого фильтра можно представить так:

$$F_{x,y} = \frac{1}{K} \sum_{i=i_{min}}^{i_{max}} \sum_{j=j_{min}}^{j_{max}} P_{x+j,y+i} \cdot M_{i-i_{min},j-j_{max}},$$

где P — значение цвета текущего пиксела, F — новое значение цвета пиксела, K — нормирующий коэффициент, M — двумерный массив коэффициентов, который определяет свойства фильтра (обычно этот массив называют маской).

Размеры окна фильтра: $(j_{max} - j_{min}+1)$ по горизонтали и $(i_{max} - i_{min}+1)$ — по вертикали. При $i_{min}, j_{min} = -1$ и $i_{max}, j_{max} = +1$ имеем фильтр с окном 3×3, который часто используется на практике.

Для обработки всего растра необходимо произвести указанные выше вычисления для каждого пиксела. Если в ходе обработки новые значения цвета

пикселов записываются в исходный растр и вовлекаются в вычисления для очередных пикселов, то такую фильтрацию называют *рекурсивной*. При *нерекурсивной* фильтрации в вычисления вовлекаются только прежние значения цвета пикселов. Нерекурсивность можно обеспечить, если новые значения записывать в отдельный массив.

На рис. 1.35 представлены результаты работы двух вариантов сглаживающего нерекурсивного фильтра с окном (маской) 3×3.

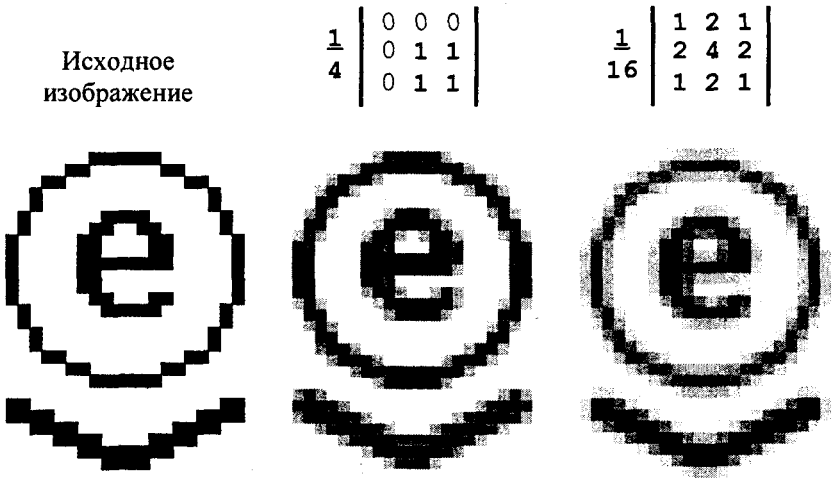


Рис. 1.35. Два сглаживающих фильтра с маской 3×3

Значение нормирующего коэффициента здесь выбрано равным сумме элементов маски. Этим обеспечивается сохранение масштаба яркости преобразованного растра. Заметьте, что маска — это не матрица, а массив коэффициентов, располагающихся соответственно пикселам окна. Средний фильтр можно задать и маской 2×2 — отбросить нулевые коэффициенты.

При сглаживании цветных изображений можно использовать модель RGB и производить фильтрацию по каждой компоненте.

С помощью локальной цифровой фильтрации можно выполнять достаточно разнообразную обработку изображений — повышение резкости, выделение контуров и многое другое [31, 34].

Дизеринг

Хорошо, когда растровое устройство отображения может прямо воссоздавать тысячи цветов для каждого пиксела. Не так давно это было проблемой даже для компьютерных дисплеев (а точнее — для видеоадаптеров). Современные

растровые дисплеи достаточно качественно отображают миллионы цветов, благодаря чему без проблем можно отображать цветные фотографии. Но для растровых устройств, которые печатают на бумаге, положение совсем другое. Устройства печати обычно имеют высокую разрешающую способность (dpi), часто на порядок большую, чем дисплеи. Однако нельзя непосредственно воссоздать даже сотню градаций серого для пикселов черно-белых фотографий, не говоря уже о миллионах цветов. Вы можете возразить, что в любой газете или журнале мы видим иллюстрации. Возьмите лупу и посмотрите, например, на изображение любой напечатанной фотографии. В большинстве случаев можно увидеть, что оттенки цветов (для цветных изображений) или полутоновые градации (для черно-белых) имитируются комбинированием, смесью точек. Чем качественнее полиграфическое оборудование, тем меньше отдельные точки и расстояние между ними.

Иногда отдельные точки на фотографии нельзя различить даже с помощью лупы, что может быть в таких случаях — или нам посчастливилось увидеть печать многими сотнями красок, или разрешающая способность устройства печати очень высокая. Оба варианта пока что не встречаются. Однако, безусловно, с течением времени будут изобретены способы печати если не многими тысячами красок (что маловероятно), то хотя бы красками, которые плавно изменяют свой цвет, или будет изобретена бумага с соответствующими свойствами [51].

Для устройств печати на бумаге проблема количества красок достаточно важна. В полиграфии для цветных изображений обычно используют три цветных краски и одну черную, что в смеси дает восемь цветов (включая черный и белый цвет бумаги). Встречаются образцы печати большим количеством красок — например, карты, напечатанные с использованием восьми красок, однако такая технология печати намного сложнее. Состояние дел с цветной печатью можно оценить также на примере относительно простых офисных принтеров. Недавно появились струйные принтеры с шестью цветными красками вместо трех. В таких принтерах в состав обычных CMYK-красок добавлены бледно-голубая, бледно-пурпурная и бледно-желтая краски (семицветные принтеры). В шестицветных принтерах отсутствует бледно-желтая краска [43]. Увеличение количества красок значительно улучшило качество печати, однако и этого пока явно мало.

Если графическое устройство не способно воссоздавать достаточное количество цветов, тогда используют растрирование — независимо от того, растровое это устройство или не растровое. В полиграфии растрирование известно давно [14, 21]. Оно использовалось несколько столетий тому назад для печати гравюр. В гравюрах изображение создается многими штрихами, причем полутоновые градации реализованы или штрихами различной толщины на

одинаковом расстоянии, или штрихами одинаковой толщины с переменной густотою расположения. Такие способы используют особенности человеческого зрения и в первую очередь — пространственную интеграцию. Если достаточно близко расположить маленькие точки различных цветов, то они будут восприниматься как одна точка с некоторым усредненным цветом. Если на плоскости густо расположить много маленьких разноцветных точек, то будет создана визуальная иллюзия закрашивания плоскости некоторым усредненным цветом. Однако если увеличивать размеры точек и (или) расстояние между ними, то иллюзия сплошного закрашивания исчезает — включается другая система человеческого зрения, обеспечивающая нашу способность различать отдельные объекты, подчеркивать контуры.

В компьютерных графических системах часто используют эти методы. Они позволяют увеличить количество оттенков цветов за счет снижения пространственного разрешения растрового изображения. Иначе говоря — это обмен разрешающей способности на количество цветов. В литературе по компьютерной графике такие методы растрирования получили название **dithering** (дрожание, разрежение).

Простейшим вариантом дизеринга можно считать создание оттенка цвета парами соседних пикселов.

Если рассмотреть ячейки из двух пикселов (рис. 1.36), то ячейка номер 1 дает оттенок цвета

$$C = (C_1 + C_2) / 2,$$

где C_1 и C_2 — цвета, которые графическое устройство способно непосредственно воспроизвести для каждого пиксела. Числовые значения C, C_1 и C_2 можно рассчитать в полутоновых градациях или в модели RGB — отдельно для каждой компоненты.

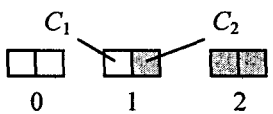


Рис. 1.36. Ячейки из двух пикселов

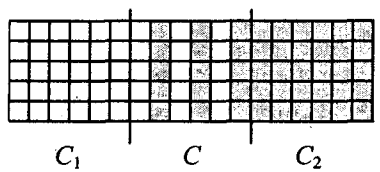


Рис. 1.37. Простейший дизеринг

Пример растра с использованием ячеек из двух пикселов приведен на рис. 1.37. Как видим, для создания промежуточного оттенка C ячейки образовывают вертикальные линии, которые очень заметны. Для того чтобы че-

ловек воспринял это как сплошной оттенок, необходимо, чтобы угловой размер ячеек был меньше одной угловой минуты. Можно изменять положения таких ячеек в растре, располагая их, например, по диагонали. Это несколько лучше, но не намного.

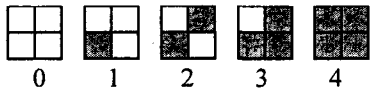


Рис.1.38. Ячейки 2×2

Чаще используют квадратные ячейки больших размеров. Дадим пример ячеек размером 2×2 (рис. 1.38).

Такие ячейки дают 5 градаций, из них три комбинации (1, 2, 3) образуют новые оттенки.

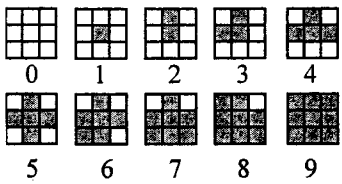


Рис. 1.39. Ячейки 3×3 представляют 10 градаций

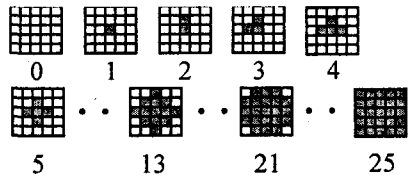
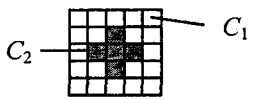


Рис. 1.40. Ячейки 5×5 дают 26 градаций

Расчет цвета, соответствующего одной из комбинаций пикселов в ячейке, можно выполнить таким образом. Если пикселы ячейки могут быть только двух цветов (C_1 и C_2), то необходимо подсчитать часть площади ячейки для пикселов каждого цвета. Цвет ячейки (C) можно оценить соотношением

$$C = \frac{S_1C_1 + (S - S_1)C_2}{S'} = \frac{S_1C_1 + S_2C_2}{S},$$

где S — общая площадь ячейки; S_1 и S_2 — части площади, занятых пикселами цветов C_1 и C_2 соответственно, причем $S_1 + S_2 = S$. Проще всего, когда пикселы квадратные, а их размер равен шагу размещения пикселов. Примем площадь одного пиксела за единицу. В этом случае площадь, занимаемая пикселами в ячейке, равна их количеству (рис. 1.41).



$S = 25, \quad S_1 = 20, \quad S_2 = 5,$
$C = (20\,C_1 + 5\,C_2) / 25.$

Рис. 1.41. Здесь площадь определяется количеством пикселов

Для ячейки 5×5, изображенной на рис. 1.41, дадим расчет цвета C для некоторых цветов C_1 и C_2. Пусть C_1 — белый цвет $(R_1G_1B_1) = (255, 255, 255)$, а C_2 — черный $(R_2G_2B_2) = (0, 0, 0)$, тогда

$$C = \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} (S_1R_1 + S_2R_2)/S \\ (S_1G_1 + S_2G_2)/S \\ (S_1B_1 + S_2B_2)/S \end{pmatrix} = \begin{pmatrix} 204 \\ 204 \\ 204 \end{pmatrix},$$

то есть мы получили светло-серый цвет.

Еще пример. Если C_1 — желтый $(R_1G_1B_1) = (255, 255, 0)$, а C_2 — красный $(R_2G_2B_2) = (255, 0, 0)$, то $C = (255, 204, 0)$. Это оттенок оранжевого цвета.

Следовательно, если в ячейке размерами $n \times n$ использованы два цвета, то с помощью этой ячейки можно получить n^2+1 различных цветовых градаций. Две комбинации пикселов — когда все пикселы ячейки имеют цвет C_1 или C_2 — дают цвет ячейки соответственно C_1 или C_2. Все иные комбинации дают оттенки, промежуточные между C_1 и C_2.

Можно считать, что ячейки размером $n \times n$ образовывают растр с разрешающей способностью в n раз меньшей, чем у исходного растра, а глубина цвета возрастает пропорционально n^2. Для характеристики изображений, которые создаются методом дизеринга, используют термин ***линиатура*** растра. Линиатура вычисляется как количество линий (ячеек) на единицу длины — сантиметр, миллиметр, дюйм. В последнем случае единицей измерения для линиатуры является **lpi** (по аналогии с **dpi**).

Как реализовать метод дизеринга в графической системе? Рассмотрим примеры преобразования растрового изображения размером $p \times q$ с определенной глубиной цвета в другой растр, предназначенный для отображения с помощью графического устройства, в котором используется ограниченное количество основных цветов. В таком случае нужно выбрать размеры ячейки $m \times n$, которые обеспечивают достаточное количество цветовых градаций. Затем каждый пиксел растра превращается в пиксел растра отображения. Это можно осуществить двумя способами.

Первый способ. Каждый пиксел заменяется ячейкой из $m \times n$ пикселов. Это самое точное преобразование по цветам, но размер растра увеличивается и равен $mp \times nq$ пикселов.

Второй способ. Размер растра в пикселах не изменяется, если пиксел растра отображения образовывается следующим образом:

1. Определяем координаты пиксела (x, y) для преобразуемого растра.
2. Находим цвет пиксела (x, y).

3. По цвету пиксела находим номер (k) ячейки, наиболее адекватно представляющей этот цвет.
4. По координатам (x,y) вычисляем координаты пиксела внутри ячейки:

   $x_k = x \bmod m$; $y_k = y \bmod n$.

5. Находим цвет (C) пиксела ячейки с координатами (x_k,y_k).
6. Записываем в растр отображения пиксел (x,y) с цветом C.

Такой способ можно использовать не для любых вариантов расположения пикселов в ячейках. Конфигурации пикселов должны быть специально разработаны для таких преобразований. Одно из требований можно сформулировать так. Если ячейки разработаны на основе двух цветов, например, белого и черного, а градации изменяются пропорционально номеру ячейки, то необходимо, чтобы ячейка с номером (i) для более темной градации серого содержала бы все черные пикселы ячейки номер ($i - 1$).

Рассмотрим пример изображения, созданного на основе ячеек 5×5.

Для создания такого изображения специально была выбрана небольшая разрешающая способность, чтобы подчеркнуть структуру изображения. Ячейки образовывают достаточно заметный квадратный растр (рис. 1.42).

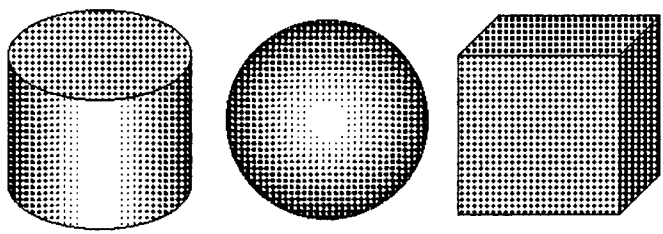


Рис. 1.42. Пример квадратного растра с ячейками 5×5

Для улучшения восприятия изображения можно использовать иное расположение ячеек, например, диагональное (рис 1.43).

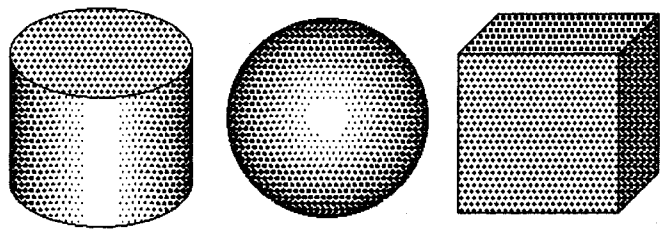


Рис. 1.43. Диагональное расположение ячеек 5×5

Диагональное расположение можно получить, если сдвигать четные строки ячеек (рис. 1.44).

Координаты пикселов ячеек можно вычислять следующим образом:

$$s = \left(\left(\frac{y}{n} \right) \bmod 2 \right) \frac{m}{2},$$

$$x_k = (x - s) \bmod m,$$

$$y_k = y \bmod n.$$

Для того чтобы получить диагональную структуру растра подобную той, что используется для печати газет, можно использовать квадратное расположение ячеек другого типа (рис. 1.45).

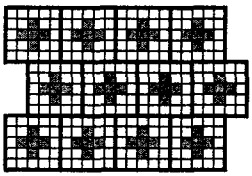


Рис. 1.44. Пример диагонального размещения ячеек

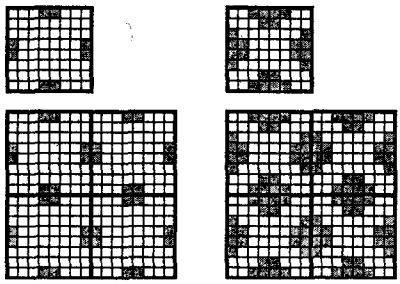


Рис. 1.45. Еще один пример диагональной структуры

Вы, наверное, уже заметили, что для всех приведенных выше примеров дизеринга ячейки образовывают точки переменного размера с постоянным шагом. Однако часто используется иной подход — переменная густота расположения точек постоянного размера. Такой способ получил название частотной модуляции (ЧМ) (рис. 1.46).

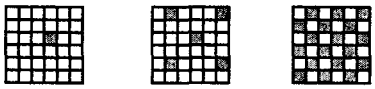


Рис. 1.46. ЧМ-ячейки 6×6

Положительная черта способа ЧМ — меньшая заметность структуры растра. Однако его использование затруднено в случае, когда размер пикселов больше, чем их шаг. Начиная с определенной густоты, пикселы смыкаются. Кроме того, на дискретном растре невозможно обеспечить плавное изменение густоты (частоты), в особенности для ячеек небольшого размера. Рассмотрим пример ячеек 5×5, реализующих ЧМ-дизеринг (рис. 1.47).

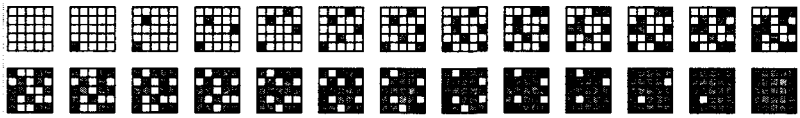


Рис. 1.47. Набор ЧМ-ячеек 5×5

Для изображений, созданных методом ЧМ-дизеринга, наблюдается меньшая заметность растровой структуры (рис. 1.48).

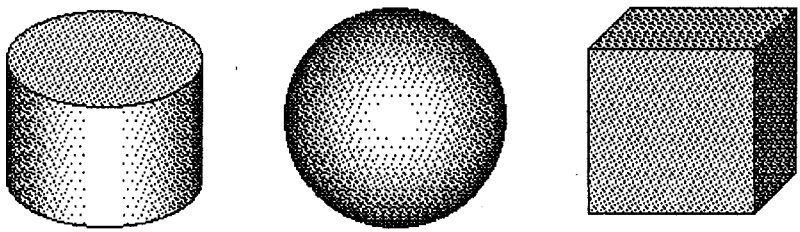


Рис. 1.48. Диагональное расположение ЧМ-ячеек 5×5

Однако при регулярном расположении одинаковых ячеек всегда образовывается текстура, муар, лишние контуры. Одна из важных задач — разработка таких вариантов ячеек, которые предопределяют наименее заметную растровую структуру (кроме тех случаев, когда, наоборот, такую структуру нужно подчеркнуть для создания изображения в стиле гравюры). Это довольно сложная задача.

Один из способов создания достаточно качественных изображений — это диффузный дизеринг (diffused dithering). Суть его в том, что ячейки создаются случайно (или псевдослучайно). Если для каждой градации создавать случайные ячейки, то даже для фрагмента растра пикселов с постоянным цветом не будут образовываться регулярные структуры. Это соответствует диффузному отражению света от матовой поверхности.

1.5. Эволюция компьютерных видеосистем

Компьютерные видеосистемы рассмотрим на примере персональных компьютеров класса IBM PC. Первый персональный компьютер фирмы IBM появился в 1981 году. Сейчас уже можно сказать, что появление как раз этого компьютера привело к значительному распространению персональных компьютеров. Некоторые особенности архитектуры IBM PC сохранены и по сей день (рис. 1.49).

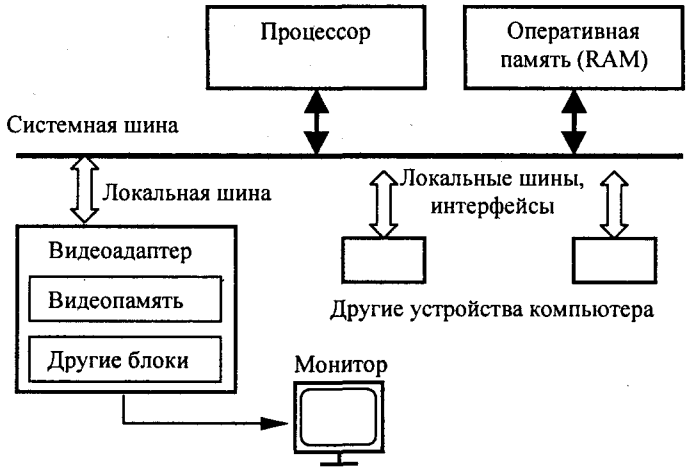


Рис. 1.49. Общая упрощенная структура персонального компьютера

Одной из таких особенностей, которые выгодно отличали его от других персональных компьютеров, является открытость архитектуры. Это означает гибкие возможности подключения разнообразных устройств, простоту модернизации компьютера. Важным фактором была цена — для IBM PC она была меньшей, чем, например, для компьютеров Apple, которые были лучше по другим показателям (в первую очередь, по графическим возможностям). Кроме того, с самого начала пользователям компьютеров семейства IBM PC были доступны разнообразные средства программирования — миллионы пользователей получили возможность сами разрабатывать программное обеспечение. Все это привело к массовому распространению таких компьютеров, использованию в разнообразных областях.

Важная черта архитектуры персонального компьютера с позиций графики — то, что контроллер видеосистемы (видеоадаптер) расположен рядом с процессором и оперативной памятью, так как подключен к системной шине через скоростную локальную шину. Это дает возможность быстро вести обмен данными между оперативной памятью и видеопамятью. Для вывода графических изображений, в особенности в режиме анимации, нужна самая высокая скорость передачи данных. В отличие от этого в больших компьютерах (мэйнфреймах) данные к дисплеям передавались через интерфейс канала ввода-вывода, который работает намного медленнее, чем системная шина. Большие компьютеры, как правило, работают со многими дисплеями, расположенными на значительном расстоянии.

Первый компьютер IBM PC был оснащен видеоадаптером MDA (Monochrome Display Adapter). Видеосистема была предназначена для работы в текстовом режиме — отображалось 25 строк по 80 символов в каждой строке.

Год спустя небольшая фирма Hercules выпустила видеоадаптер Hercules Graphic Card. Он поддерживал также и графический черно-белый режим 720×348.

Следующим шагом был видеоадаптер CGA (Color Graphic Adapter). Это первая цветная модель для IBM PC. Адаптер CGA позволял работать в цветном текстовом или графическом режимах. Далее мы будем рассматривать только графические режимы видеоадаптеров. Графических режимов для CGA было два: черно-белый 640×200 и цветной 320×200. В цветном режиме можно было отображать только четыре цвета одновременно (2 бита на пиксел).

В 1984 году появился адаптер EGA (Enhanced Graphic Adapter). Это было значительное достижение для персональных компьютеров этого типа. Появился графический 16-цветный видеорежим 640×350 пикселов. Цвета можно выбирать из палитры 64 цветов. В это время уже получили распространение компьютерные игры с более или менее качественной графикой и графические программы для работы. Однако шестнадцати цветов явно мало для показа изображений типа фотографий, а разрешающая способность недостаточна для графических пакетов типа САПР. Кроме того, видеорежим 640×350 имеет еще один недостаток — различная разрешающая способность по горизонтали и вертикали — "не квадратные пикселы".

В 1987 году появились видеоадаптеры MCGA (Multi-Color Graphic Array) и VGA (Video Graphic Array). Они обеспечивали уже 256-цветные видеорежимы.

Более совершенным был адаптер VGA — он стал наиболее популярным. Адаптер VGA имел 256-цветный графический видеорежим с размерами растра 320×200. Цвета можно выбирать из палитры в 256 тысяч цветов. Это дало возможность полностью удовлетворить потребности отображения полутоновых черно-белых фотографий. Цветные фотографии отображались достаточно качественно, однако 256 цветов мало, поэтому в компьютерных играх и графических пакетах активно использовался дизеринг. Кроме того, режим 320×200 тоже имеет различную разрешающую способность по горизонтали и вертикали. Для мониторов, которые используются в персональных компьютерах типа IBM PC, необходимо, чтобы количество пикселов по горизонтали и вертикали была в пропорции 4:3. То есть, не 320×200, а 320×240. Такого документированного видеорежима для VGA нет, однако в литературе [39] приведен пример, как создать 256-цветный видеорежим 320×240 на видеосистеме VGA. Можно запрограммировать видеоадаптер, записав в его регистры соответствующие значения, и получить видеорежим "X" (не путать с XGA). Несколько лет тому назад автор этой книги проверял сведения, приведенные в статье [39], и может подтвердить, что действительно, "X — видео-

режим" 256 цветов 320×240 может быть установлен для любой видеосистемы VGA (во всяком случае, не было найдено ни одной, где этого делать нельзя).

Видеоадаптер VGA также имеет 16-цветный видеорежим 640×480. Это соответствует "квадратным пикселам". Рост разрешающей способности в сравнении с EGA не очень большой, но ощутимый, что дает новый толчок для развития графических программ на персональных компьютерах.

Дальнейшее развитие видеоадаптеров для компьютеров типа IBM PC связано с повышением разрешающей способности и количества цветов. Можно отметить видеосистему IBM 8514, которая была предназначена для работы с пакетами САПР. Стали появляться видеоадаптеры различных фирм, которые обеспечивали на первых порах видеорежимы 800×600, а потом и 1024×768 цветов при 16-ти цветах, и видеорежимы 640×480, 800×600 и больше для 256 цветов. Эти видеоадаптеры стали называть SuperVGA. Чуть позже появился видеоадаптер IBM XGA.

Первой достигла глубины цвета в 24 бит фирма Targa с видеоадаптером Targa24, что позволило получить на персональных компьютерах IBM PC видеорежим True Color. Такое достижение можно считать началом профессиональной графики на персональных компьютерах этого типа. Там, где ранее использовались графические рабочие станции или персональные компьютеры Apple Macintosh, отныне постепенно переходили на более дешевые компьютеры IBM PC. Одной из таких областей является компьютерное настольное издательство.

Сейчас на компьютерах IBM PC с процессором Pentium используется много типов видеоадаптеров. Все видеосистемы растрового типа. Некоторые из них позволяют устанавливать глубину цвета 32 бит на пиксел при размерах растра 1600×1200 и более. Существуют стандарты на видеорежимы, установленные VESA (Video Electronic Standards Association).

Параметры отображения обусловливаются не только моделью видеоадаптера, но и объемом установленной видеопамяти. Видеопамять персонального компьютера (VRAM — Video RAM) хранит растровое изображение, которое показывается на экране монитора. Изображение на мониторе полностью соответствует текущему содержанию видеопамяти. Видеопамять постоянно сканируется с частотой кадров монитора. Запись новых данных в видеопамять немедленно изменяет изображение на мониторе. Необходимый объем видеопамяти вычисляется как площадь растра экрана в пикселах, умноженная на количество бит (или байтов) на пиксел. Например, для 24-битного видеорежима 1024×768 нужна видеопамять 24×1024×768 = 18 874 368 бит = = 2.25 Мбайт.

В видеоадаптерах первых образцов количество видеопамяти вычислялось в килобайтах, например, адаптер CGA имел 16 Кбайт [6, 29]. В современных видеоадаптерах счет идет на мегабайты. Как правило, объем видеопамяти кратен степени двойки — 1, 2, 4, 8 Мбайт, встречаются также видеоадаптеры с 16 Мбайт и более. Наблюдается тенденция увеличения объемов видеопамяти — соответственно увеличению разрешающей способности и глубины цвета видеосистем. В видеопамяти могут храниться несколько кадров изображения — это может быть использовано при анимации. Кроме того, в некоторых видеоадаптерах предусмотрена возможность использования видеопамяти для хранения другой информации, например Z-буфера, растров текстур.

Адреса, по которым процессор обращается к видеопамяти, находятся в общем адресном пространстве. Например, для видеорежимов VGA 256 цветов 320×200, 16 цветов 640×480, а также для некоторых других, адрес первого байта видеопамяти равен A000:0000 (сегмент : смещение) или A0000 (абсолютный адрес) (рис. 1.50).

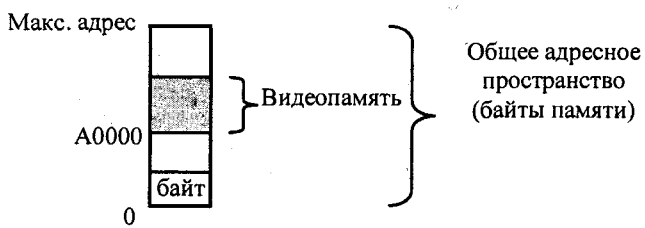


Рис. 1.50. Адрес видеопамяти для графических видеорежимов VGA

Для некоторых видеорежимов (старых образцов) используется другой адрес, например, B800:0000 для CGA 320×200. Современные видеоадаптеры обычно поддерживают видеорежимы, которые использовались ранее. Это делается для обеспечения возможности функционирования старых программ. Каждый видеорежим имеет собственный номер (код).

Кроме физической организации памяти компьютера — в виде одномерного вектора байтов в общем адресном пространстве, необходимо учитывать логическую организацию видеопамяти. Следует отметить, что названия "физическая" и "логическая" организация могут означать разные вещи для различных уровней рассмотрения. Например, если говорить о физической организации памяти, то она в микросхемах выглядит совсем не как одномерный вектор байтов, а как матрица бит. Логическая организация видеопамяти зависит от видеорежима. В качестве примера на рис. 1.51 приведен видеорежим VGA 256 цветов 320×200 (его код 13h).

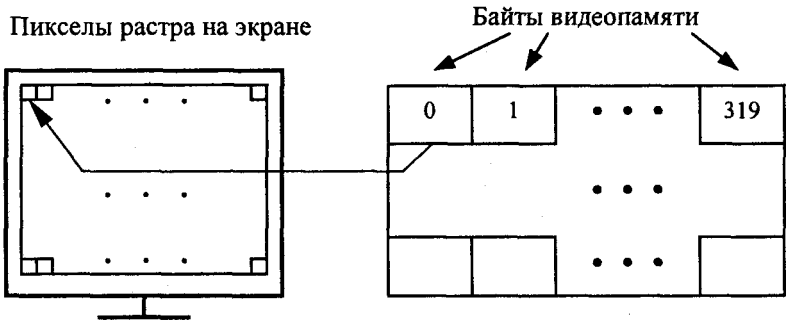


Рис. 1.51. Один байт на пиксел для видеорежима VGA 320×200

Намного сложнее логическая организация видеопамяти для видеорежима VGA 16 цветов 640×480 (код 12h), показанная на рис 1.52.

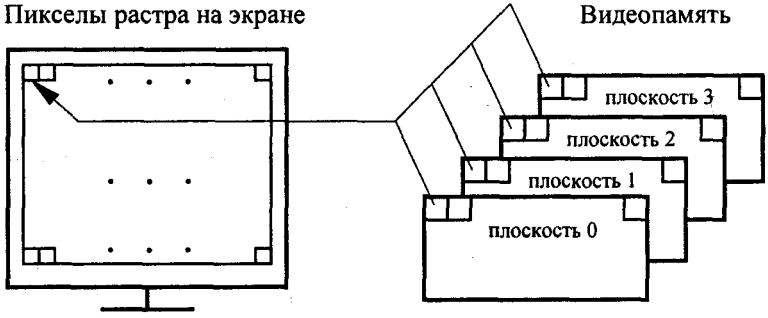


Рис. 1.52. Четыре битовые плоскости видеорежима 16 цвета 640×480

В этом видеорежиме используются четыре массива байтов памяти. Каждый массив назван битовой плоскостью, для каждого пиксела используются одинаковые биты данных различных плоскостей. Каждая битовая плоскость имеет 80 байтов в одной строке. Плоскости имеют одинаковый адрес в памяти, для доступа к отдельной плоскости необходимо устанавливать индекс плоскости в соответствующем регистре видеоадаптера. Подобный способ организации видеопамяти используется во многих других видеорежимах, он позволяет, например, быстро копировать массивы пикселов.

Для сохранения нескольких кадров изображения в некоторых видеорежимах предусматриваются отдельные страницы видеопамяти с одинаковой логической организацией. Тогда можно изменять стартовый адрес видеопамяти — это приводит к сдвигу изображения на экране. Во всех графических видеорежимах стартовый адрес видеопамяти соответствует левому верхнему пикселу на экране. Поэтому координатная система с центром координат (0,0)

в левом верхнем углу растра часто используется в качестве основной (или устанавливается по умолчанию) во многих графических интерфейсах программирования, например, в API Windows.

Обмен данными по системной шине для видеосистемы обеспечивают процессор, видеоадаптер и контроллер локальной шины. До недавнего времени для подключения видеоадаптеров использовалась локальная шина PCI (Peripheral Component Interconnect local bus). Шина PCI предназначена не только для графики, она является стандартом подключения самых разнообразных устройств, например, модемов, сетевых контроллеров, контроллеров интерфейсов. Эта шина — 32-битная, работает на частоте 33 МГц, скорость обмена до 132 Мбайт/с.

В настоящее время видеоадаптеры подключаются через локальную шину AGP (Accelerated Graphics Port). Разрядность — 64 бит. На частоте 66 МГц обеспечивала скорость обмена 528 Мбайт/с. Сейчас AGP работает и на более высоких частотах. Шина AGP была разработана для повышения скорости обмена данными между видеоадаптером и оперативной памятью в сравнении с возможностями шины PCI. Это позволяет достичь большей частоты кадров при работе графических 3D-акселераторов. Высокая скорость обмена с оперативной памятью также позволяет хранить в этой памяти растровые текстуры (ранее для этого часто использовалась видеопамять, однако она обычно имеет недостаточную для этого емкость). Наличие AGP-порта также приводит к возрастанию быстродействия компьютера в целом благодаря уменьшению нагрузки на шину PCI, что дает возможность эффективнее использовать последнюю для работы с сетью, мультимедиа.

Современные видеоадаптеры представляют собой сложные электронные устройства. Кроме видеопамяти, на плате видеоадаптера (сейчас его часто называют видеокартой) располагается мощный специализированный графический процессор, который по сложности уже приближается к центральному процессору. Кроме визуализации содержимого видеопамяти графический процессор видеоадаптера выполняет как относительно простые растровые операции — копирование массивов пикселов, манипуляции с цветами пикселов, так и более сложные. Там, где ранее использовался исключительно центральный процессор, в данное время все чаще применяется графический процессор видеоадаптера, например, для выполнения операций графического вывода линий, полигонов. Первые графические процессоры видеоадаптеров выполняли преимущественно операции рисования плоских элементов. Современные графические процессоры выполняют уже много базовых операций 3D-графики, например, поддержку Z-буфера, наложение текстур и тому подобное. Видеоадаптер выполняет эти операции аппаратно, что позволяет намного ускорить их в сравнении с программной реализацией данных опера-

ций центральным процессором. Так появился термин **графические акселераторы**. Быстродействие таких видеоадаптеров часто измеряется в количестве графических элементов, которые рисуются за одну секунду. Современные графические акселераторы способны рисовать миллионы треугольников за секунду. Возможности графических акселераторов сейчас активно используются разработчиками компьютерных игр.

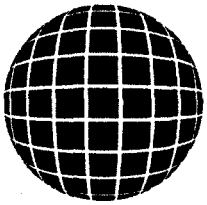
Широка номенклатура видеоадаптеров для персональных компьютеров. Несколько примеров. Видеоадаптеры Matrox (качественная двумерная графика), NVidia GeForce (игровые 3D-акселераторы), 3Dlabs Wildcat (для профессионального 3D-моделирования).

Использование программистами графических возможностей видеосистемы может осуществляться различно. Во-первых, простейшие операции, такие как определение графического видеорежима, вывод пиксела на экран и некоторые другие, поддерживаются BIOS. Во-вторых, можно использовать функции операционной системы. Различные операционные системы могут предоставлять различные возможности. Например, в MS-DOS графических функций почти не было, однако программисту был разрешен свободный доступ ко всем аппаратным ресурсам компьютера. В быстродействующих графических программах часто использовался непосредственный доступ к видеопамяти. В отличие от этого, операционная система Windows запрещает прикладным программам непосредственный доступ к аппаратным ресурсам, однако можно применять несколько сотен графических функций операционной системы — интерфейс API. В-третьих, можно использовать специализированные графические интерфейсы, которые поддерживают аппаратные возможности современных графических процессоров.

Одним из наиболее известных графических интерфейсов является OpenGL. Этот интерфейс в виде библиотеки графических функций был разработан Silicon Graphics, и поддерживается многими операционными системами (в том числе Windows), а также производителями графических акселераторов. Интерфейс OpenGL для графического отображения использует взаимодействие типа клиент-сервер [25, 61].

Другим известным графическим интерфейсом является DirectX с подсистемой трехмерной графики Direct3D, а также подсистемой Direct Draw, которая обеспечивает, в частности, непосредственный доступ к видеопамяти. Этот интерфейс разработан Microsoft и предназначен только для Windows [26].

Известны также другие разработки графических интерфейсов для видеоадаптеров. Например, интерфейс GLide, разработанный 3Dfx для графических видеоадаптеров семейства Voodoo (впрочем, это уже история, поскольку фирма 3Dfx недавно прекратила свое существование как изготовитель видеоадаптеров).



ГЛАВА 2

Координатный метод

Координатный метод был введен в XVII веке французскими математиками Р. Декартом и П. Ферма. На этом методе зиждется *аналитическая геометрия*, которую можно считать фундаментом компьютерной графики. В современной компьютерной графике широко используется координатный метод. Этому есть несколько причин.

- Каждая точка на экране (или на бумаге при печати на принтере) задается координатами — например, пикселными.
- Координаты используются для описания объектов, которые будут отображаться как пространственные. Например, объекты микромира, объекты на поверхности Земли, объекты космического пространства и тому подобное. Даже тогда, когда отображается нечто, не имеющее привязки к положению в пространстве (например, случайные цветовые пятна в каком-то видеоэффекте), то и здесь используются координаты для учета взаиморасположения отдельных элементов.
- При выполнении многих промежуточных действий отображения используют разные системы координат и преобразования из одной системы в другую.

2.1. Преобразование координат

Сначала рассмотрим общие вопросы преобразования координат. Пусть задана n-мерная система координат в базисе $(k_1, k_2, ..., k_n)$, описывающая положение точки в пространстве с помощью числовых значений k_i. В компьютерной графике чаще всего используется двумерная (n=2) и трехмерная (n=3) системы координат.

Если задать другую, N-мерную, систему координат в базисе $(m_1, m_2, ..., m_N)$ и поставить задачу определения координат в новой системе, зная координаты в старой, то решение (если оно существует) можно записать в таком виде:

$$\begin{cases} m_1 = f_1(k_1, k_2, ..., k_n), \\ m_2 = f_2(k_1, k_2, ..., k_n), \\ \cdots \\ m_N = f_N(k_1, k_2, ..., k_n), \end{cases}$$

где f_i — функция пересчета i-й координаты, аргументы — координаты в системе k_i.

Можно поставить и обратную задачу — по известным координатам $(m_1, m_2, ..., m_N)$ определить координаты $(k_1, k_2, ..., k_n)$. Решение обратной задачи запишем так:

$$\begin{cases} k_1 = F_1(m_1, m_2, ..., m_N), \\ k_2 = F_2(m_1, m_2, ..., m_N), \\ \cdots \\ k_n = F_n(m_1, m_2, ..., m_N), \end{cases}$$

где F_i — функции *обратного* преобразования.

В случае, когда размерности систем координат не совпадают ($n \neq N$), осуществить однозначное преобразование координат зачастую не удается. Например, по двумерным экранным координатам нельзя без дополнительных условий однозначно определить трехмерные координаты отображаемых объектов.

Если размерности систем совпадают ($n = N$), то также возможны случаи, когда нельзя однозначно решить прямую или обратную задачи.

Преобразование координат классифицируют:

- по системам координат — например, преобразование из полярной системы в прямоугольную;
- по виду функций преобразования f_i.

По виду функций преобразования различают *линейные* и *нелинейные* преобразования. Если при всех $i = 1, 2, ..., N$ функции f_i — линейные относительно аргументов $(k_1, k_2, ..., k_n)$, то есть

$$f_i = a_{i1}k_1 + a_{i2}k_2 + ... + a_{in}k_n + a_{i,n+1},$$

где a_{ij} — константы, то такие преобразования называются линейными, а при $n = N$ — *аффинными*.

Если хотя бы для одного i функция f_i — нелинейная относительно $(k_1, k_2, ..., k_n)$, тогда преобразование координат в целом не линейно.

Например, преобразование

$$X = 3x + 5y,$$
$$Y = 4xy + 10y$$

нелинейное, так как в выражении для Y присутствует xy.

Тем, кто интересуется математическими аспектами, относящимся к системам координат и преобразованиям систем координат, можно порекомендовать такие книги, как [16, 23].

Линейные преобразования наглядно записываются в матричной форме:

$$\begin{pmatrix} m_1 \\ m_2 \\ \cdot \\ \cdot \\ \cdot \\ m_N \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \ldots & a_{2n} & a_{2,n+1} \\ & & \cdot\ \cdot\ \cdot & & \\ a_{N1} & a_{N2} & \ldots & a_{Nn} & a_{N,n+1} \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ \\ \\ k_N \end{pmatrix}.$$

Здесь матрица коэффициентов (a_{ij}) умножается на матрицу-столбец (k_i) и в результате получается матрица-столбец (m_i).

Мы и далее часто будем использовать умножение матриц, поэтому сделаем небольшой экскурс в матричную алгебру. Для двух матриц A размером $(m \times n)$ и B размером $(n \times p)$:

$$A = \begin{pmatrix} a_{11}\, a_{12} \ldots\ a_{1n} \\ a_{21}\, a_{22} \ldots\ a_{2n} \\ - \ - \ - \ - \ - \\ a_{m1}\, a_{m2} \ldots\ a_{mn} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11}\, b_{12} \ldots\ b_{1p} \\ b_{21}\, b_{22} \ldots\ b_{2p} \\ - \ - \ - \ - \ - \\ b_{n1}\, b_{n2} \ldots\ b_{np} \end{pmatrix},$$

произведением матриц является матрица $C = A\,B$ размером $(m \times p)$

$$C = \begin{pmatrix} c_{11}\, c_{12} \ldots\ c_{1p} \\ c_{21}\, c_{22} \ldots\ c_{2p} \\ - \ - \ - \ - \ - \\ c_{m1}\, c_{m2} \ldots\ c_{mp} \end{pmatrix},$$

для которой элементы c_{ij} вычисляются по формуле

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Правило вычисления элементов матрицы C можно легко запомнить по названию "строка на столбец". И действительно, для вычисления любого элемента c_{ij} необходимо умножить элементы i-й строки матрицы A на элементы j-го столбца матрицы B.

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & c_{ij} & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ a_{i1} & a_{i2} & \cdot & a_{in} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & b_{1j} & \cdot \\ \cdot & \cdot & b_{2j} & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & b_{nj} & \cdot \end{pmatrix}.$$

Произведение матриц определено только для случая, когда количество столбцов матрицы A равно количеству строк матрицы B.

Подробнее с матрицами вы можете ознакомиться в специальной математической литературе, например в [5].

Ну а теперь вернемся к преобразованиям координат. Рассмотрим более подробно некоторые отдельные типы преобразований.

Аффинные преобразования на плоскости

Зададим некоторую двумерную систему координат (x, y). Аффинное преобразование координат (x, y) описывается формулами

$$X = Ax + By + C,$$
$$Y = Dx + Ey + F,$$

где $A, B, ..., F$ — константы. Значения (X, Y) можно трактовать как координаты в новой системе координат.

Обратное преобразование (X, Y) в (x, y) также является аффинным:

$$x = A'X' + B'Y + C',$$
$$y = D'X + E'Y + F'.$$

Аффинное преобразование удобно записывать в матричном виде. Константы $A, B, ..., F$ образовывают матрицу преобразования, которая, будучи умноженная на матрицу-столбец координат (x, y), дает матрицу-столбец (X, Y). Однако для того, чтобы учесть константы C и F, необходимо перейти к так называемым *однородным координатам* — добавим строку с единицами в матрицах координат:

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Матричная запись дает возможность наглядно описывать несколько преобразований, которые идут одно за другим. Например, если необходимо сначала выполнить преобразования

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} & A & \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

а потом — другое преобразование

$$\begin{pmatrix} x'' \\ y'' \\ 1 \end{pmatrix} = \begin{bmatrix} & B & \end{bmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix},$$

то это можно описать как

$$\begin{pmatrix} X'' \\ Y'' \\ 1 \end{pmatrix} = \begin{bmatrix} & B & \end{bmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} & B & \end{bmatrix} \begin{bmatrix} & A & \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Однако вместо двух преобразований можно выполнить только одно

$$\begin{pmatrix} x'' \\ y'' \\ 1 \end{pmatrix} = \begin{bmatrix} & C & \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

где матрица (C) равна произведению $(B)(A)$.

Перемножение матриц выполняется так, как это принято в линейной алгебре.

Рассмотрим частные случаи аффинного преобразования.

1. **Параллельный сдвиг** координат (рис. 2.1).

$$\begin{cases} X = x - dx, \\ Y = y - dy. \end{cases}$$

В матричной форме:

$$\begin{pmatrix} 1 & 0 & -dx \\ 0 & 1 & -dy \\ 0 & 0 & 1 \end{pmatrix}.$$

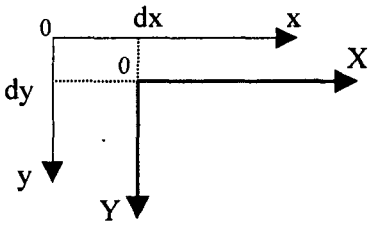


Рис. 2.1. Параллельный сдвиг координат

Обратное преобразование:

$$\begin{cases} x = X + dx, \\ y = Y + dy, \end{cases} \qquad \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}.$$

2. **Растяжение-сжатие** осей координат (рис. 2.2).

$$\begin{cases} X = x/k_x, \\ Y = y/k_y, \end{cases}$$

$$\begin{bmatrix} 1/k_x & 0 & 0 \\ 0 & 1/k_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

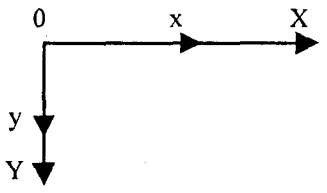


Рис. 2.2. Растяжение/сжатие осей координат

Обратное преобразование:

$$\begin{cases} x = X k_x, \\ y = Y k_y, \end{cases} \qquad \begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Коэффициенты k_x и k_y могут быть отрицательными. Например, $k_x = -1$ соответствует зеркальному отражению относительно оси y.

3. **Поворот** (рис. 2.3).

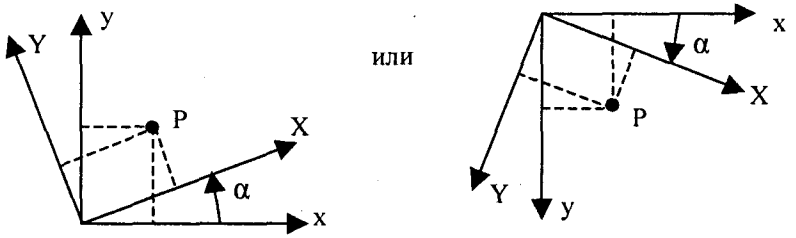


Рис. 2.3. Поворот

$$\begin{cases} X = x\cos\alpha + y\sin\alpha, \\ Y = -x\sin\alpha + y\cos\alpha, \end{cases} \qquad \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Обратное преобразование соответствует повороту системы (X, Y) на угол $(-\alpha)$.

$$\begin{cases} x = X\cos\alpha - Y\sin\alpha, \\ y = X\sin\alpha + Y\cos\alpha, \end{cases} \qquad \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Свойства аффинного преобразования.

- Любое аффинное преобразование может быть представлено как последовательность операций из числа указанных простейших: сдвиг, растяжение/сжатие и поворот.
- Сохраняются прямые линии, параллельность прямых, отношение длин отрезков, лежащих на одной прямой, и отношение площадей фигур.

Трехмерное аффинное преобразование

Запишем в виде формулы:

$$\begin{cases} X = Ax + By + Cz + D, \\ Y = Ex + Fy + Gz + H, \\ Z = Kx + Ly + Mz + N, \end{cases}$$

где $A, B, ..., N$ — константы.

Дадим также в матричной форме:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} A & B & C & D \\ E & F & G & H \\ K & L & M & N \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Для трехмерного пространства любое аффинное преобразование также может быть представлено последовательностью простейших операций. Рассмотрим их.

1. **Сдвиг** осей координат соответственно на dx, dy, dz:

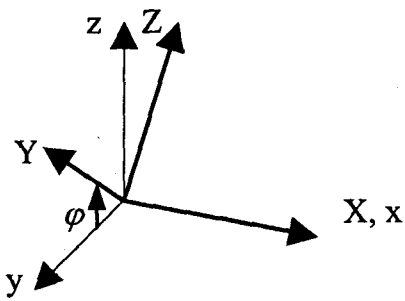
$$\begin{cases} X = x - dx, \\ Y = y - dy, \\ Z = z - dz, \end{cases} \qquad \begin{pmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

2. **Растяжение-сжатие** на k_x, k_y, k_z:

$$\begin{cases} X = x / k_x, \\ Y = y / k_y, \\ Z = z / k_z, \end{cases} \qquad \begin{pmatrix} 1/k_x & 0 & 0 & 0 \\ 0 & 1/k_y & 0 & 0 \\ 0 & 0 & 1/k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

3. **Повороты.**

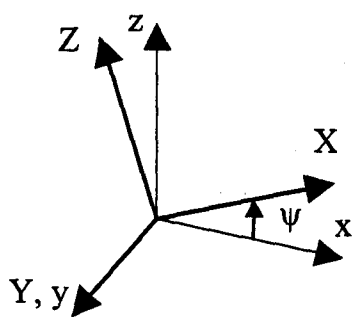
- Поворот вокруг оси x на угол φ (рис. 2.4):



$$\begin{cases} X = x, \\ Y = y\cos\varphi + z\sin\varphi, \\ Z = -y\sin\varphi + z\cos\varphi, \end{cases} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Рис. 2.4. Поворот вокруг оси на угол φ

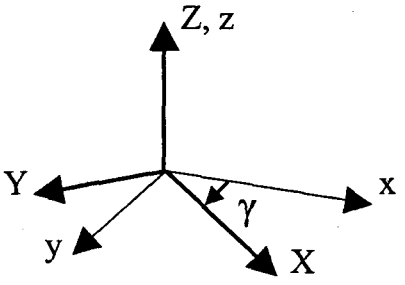
- Поворот вокруг оси y на угол ψ (рис. 2.5):



$$\begin{cases} X = x\cos\psi + z\sin\psi, \\ Y = y, \\ Z = -x\sin\psi + z\cos\psi, \end{cases} \qquad \begin{pmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Рис. 2.5. Поворот вокруг оси y на угол ψ

- Поворот вокруг оси z на угол γ (рис. 2.6):



$$\begin{cases} X = x\cos\gamma + y\sin\gamma, \\ Y = -x\sin\gamma + y\cos\gamma, \\ Z = z, \end{cases}$$

$$\begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Рис. 2.6. Поворот вокруг оси z на угол γ

2.2. Преобразование объектов

Преобразование объектов можно описать так. Пусть любая точка, которая принадлежит определенному объекту, имеет координаты $(k_1, k_2, ..., k_n)$ в n-мерной системе координат. Тогда преобразование объекта можно определить как изменение положения точек объекта. Новое положение точки пространства отвечает новым значениям координат $(m_1, m_2, ..., m_n)$.

Соотношение между старыми и новыми координатами для всех точек объекта $(m_1, m_2, ..., m_n) = F(k_1, k_2, ..., k_n)$ и будет определять преобразование объекта, где F — функция преобразования.

Классифицировать преобразования объектов можно согласно типу функции преобразования и типу системы координат.

Например, преобразование объектов на плоскости можно определить так:

$$X = F_x(x, y),$$
$$Y = F_y(x, y).$$

В трехмерном пространстве:

$$X = F_x(x, y, z),$$
$$Y = F_y(x, y, z),$$
$$Z = F_z(x, y, z).$$

Рассмотрим отдельные типы преобразований объектов.

Аффинные преобразования объектов на плоскости

Аффинные преобразования объектов на плоскости описываются так:

$$X = Ax + By + C,$$
$$Y = Dx + Ey + F,$$

где $A, B, ..., F$ — константы; x, y — координаты до преобразования; X, Y — новые координаты точек объектов.

Рассмотрим частные случаи аффинного преобразования.

1. **Сдвиг** (рис. 2.7).

$$\begin{cases} X = x + dx, \\ Y = y + dy. \end{cases}$$

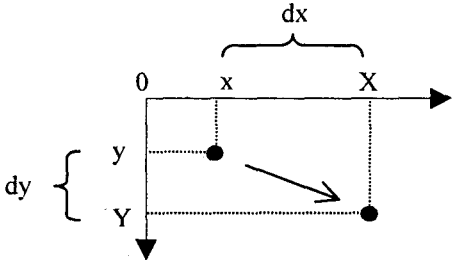


Рис. 2.7. Сдвиг

В матричной форме:

$$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix}.$$

Обратное преобразование позволяет рассчитать старые координаты точек объектов по известным новым координатам:

$$\begin{cases} x = X - dx, \\ y = Y - dy, \end{cases} \qquad \begin{pmatrix} 1 & 0 & -dx \\ 0 & 1 & -dy \\ 0 & 0 & 1 \end{pmatrix}.$$

2. **Растяжение-сжатие** (рис. 2.8).

Необходимо отметить, что это, вероятно, не очень удачное название, так как для некоторых типов объектов размеры и форма не изменяются — например, для точечных объектов. По-другому это преобразование можно назвать масштабированием.

$$\begin{cases} X = k_x x, \\ Y = k_y y. \end{cases}$$

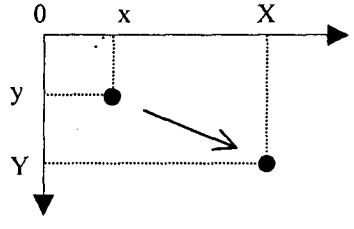


Рис. 2.8. Растяжение/сжатие

В матричной форме:

$$\begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Обратное преобразование:

$$\begin{cases} x = X/k_x, \\ y = Y/k_y, \end{cases} \qquad \begin{bmatrix} 1/k_x & 0 & 0 \\ 0 & 1/k_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. **Поворот** вокруг центра координат (0, 0) (рис. 2.9).

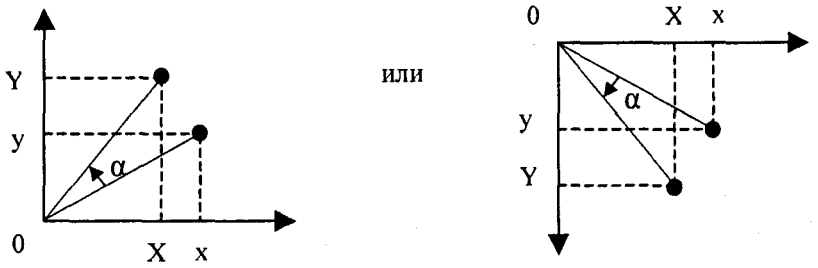


Рис. 2.9. Поворот объекта

$$\begin{cases} X = x\cos\alpha - y\sin\alpha, \\ Y = x\sin\alpha + y\cos\alpha, \end{cases} \qquad \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Формулы для обратного преобразования можно получить, если представить себе поворот точки с координатами (X, Y) на угол $(-\alpha)$:

$$\begin{cases} x = X\cos\alpha + Y\sin\alpha, \\ y = -X\sin\alpha + Y\cos\alpha, \end{cases} \qquad \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Трехмерное аффинное преобразование объектов

Приведем в виде формулы:

$$X = Ax + By + Cz + D,$$
$$Y = Ex + Fy + Gz + H,$$
$$Z = Kx + Ly + Mz + N,$$

где $A, B, ..., N$ — константы.

Рассмотрим частные случаи трехмерного аффинного преобразования объектов.

1. **Сдвиг** на dx, dy, dz.

$$\begin{cases} X = x + dx, \\ Y = y + dy, \\ Z = z + dz, \end{cases} \qquad \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

2. **Растяжение-сжатие** на k_x, k_y, k_z:

$$\begin{cases} X = k_x x, \\ Y = k_y y, \\ Z = k_z z, \end{cases} \qquad \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

3. **Повороты.**

- Поворот вокруг оси x на угол φ (рис. 2.10):

$$\begin{cases} X = x, \\ Y = y\cos\varphi - z\sin\varphi, \\ Z = y\sin\varphi + z\cos\varphi, \end{cases}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

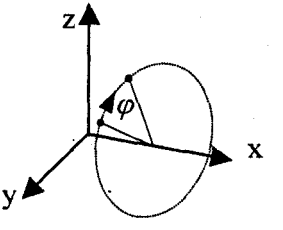


Рис. 2.10. Поворот вокруг оси x на угол φ

- Поворот вокруг оси y на угол ψ (рис. 2.11):

$$\begin{cases} X = x\cos\psi - z\sin\psi, \\ Y = y, \\ Z = x\sin\psi + z\cos\psi, \end{cases}$$

$$\begin{bmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

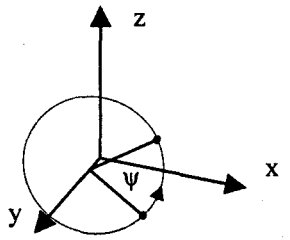


Рис. 2.11. Поворот вокруг оси y на угол ψ

- Поворот вокруг оси z на угол γ (рис. 2.12):

$$\begin{cases} X = x\cos\gamma - y\sin\gamma, \\ Y = x\sin\gamma + y\cos\gamma, \\ Z = z, \end{cases}$$

$$\begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

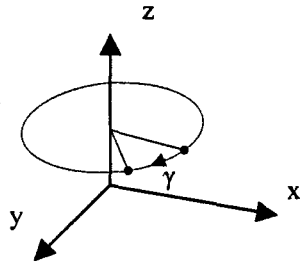


Рис. 2.12. Поворот вокруг оси z на угол γ

Формулы для обратных преобразований не приведены, но их несложно получить по аналогии с преобразованиями, которые рассмотрены выше.

2.3. Связь преобразований объектов с преобразованиями координат

Когда пользователь графической системы видит на экране перемещающийся объект, то, как вы считаете, что на самом деле происходит — перемещаются объекты или система координат в обратном направлении? Например, если в кино вы видите объекты, вращающиеся на экране по часовой стрелке, то может в действительности это камера поворачивается против часовой стрелки?

Преобразование объектов и преобразование систем координат тесно связаны между собой. Движение объектов можно рассматривать как движение в обратном направлении соответствующей системы координат.

Такая относительность для объектов отображения и систем координат дает разработчикам компьютерных систем дополнительные возможности для моделирования и визуализации пространственных объектов. С каждым объектом можно связывать как собственную локальную систему координат, так и единую для нескольких объектов. Это можно использовать, например, для моделирования подвижных объектов.

Обычно, того же самого эффекта можно добиться, если использовать различные подходы. Однако в одних случаях удобнее использовать преобразование координат, а в других — преобразование объектов. Не последнюю роль играет сложность обоснования какого-то способа, его понятность.

Рассмотрим пример комбинированного подхода. Пусть нам нужно получить функцию расчета координат (X, Y) для поворота вокруг центра с координатами (x_0, y_0) (рис. 2.13).

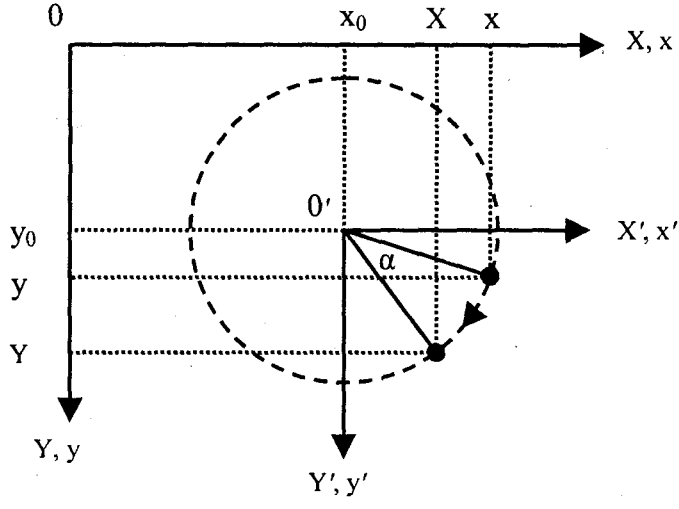


Рис. 2.13. Вращение вокруг произвольного центра

Выше мы рассмотрели поворот относительно центра координат (0, 0). Для решения нашей задачи введем новую систему координат $(x', 0', y')$ с центром в точке (x_0, y_0):

$$\begin{cases} x' = x - x_0, \\ y' = y - y_0. \end{cases}$$

Для такой системы поворот точек происходит вокруг ее центра:

$$\begin{cases} X' = x' \cos\alpha - y' \sin\alpha, \\ Y' = x' \sin\alpha + y' \cos\alpha. \end{cases}$$

Преобразуем координаты (X', Y') в (X, Y) сдвигом системы координат в точку $(0, 0)$:

$$\begin{cases} X = X' + x_0, \\ Y = Y' + y_0. \end{cases}$$

Если объединить формулы преобразований, то получим результат:

$$\begin{cases} X = (x - x_0)\cos\alpha - (y - y_0)\sin\alpha + x_0, \\ Y = (x - x_0)\sin\alpha + (y - y_0)\cos\alpha + y_0. \end{cases}$$

Решение этой задачи можно было бы осуществить и в матричной форме:

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} \text{Сдвиг системы} \\ \text{координат на} \\ -x_0, -y_0 \end{bmatrix} \begin{bmatrix} \text{Поворот} \\ \text{на угол } \alpha \end{bmatrix} \begin{bmatrix} \text{Сдвиг системы} \\ \text{координат} \\ \text{на } x_0, y_0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} =$$

$$= \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} =$$

$$= \begin{pmatrix} \cos\alpha & -\sin\alpha & -x_0\cos\alpha + y_0\sin\alpha + x_0 \\ \sin\alpha & \cos\alpha & -x_0\sin\alpha - y_0\cos\alpha + y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Рассмотрим второй пример. Нашей задачей будет вывод формул параметрического описания поверхности тора. Изобразим тор следующим образом (рис. 2.14).

Для произвольной точки P, лежащей на поверхности тора, требуется выразить координаты (x, y, z) через константы, описывающие размеры фигуры, а также через некоторые параметры. Для поверхности в трехмерном пространстве необходимо использовать два параметра. В качестве таковых выберем угловые величины: φ (широта) и ω (долгота).

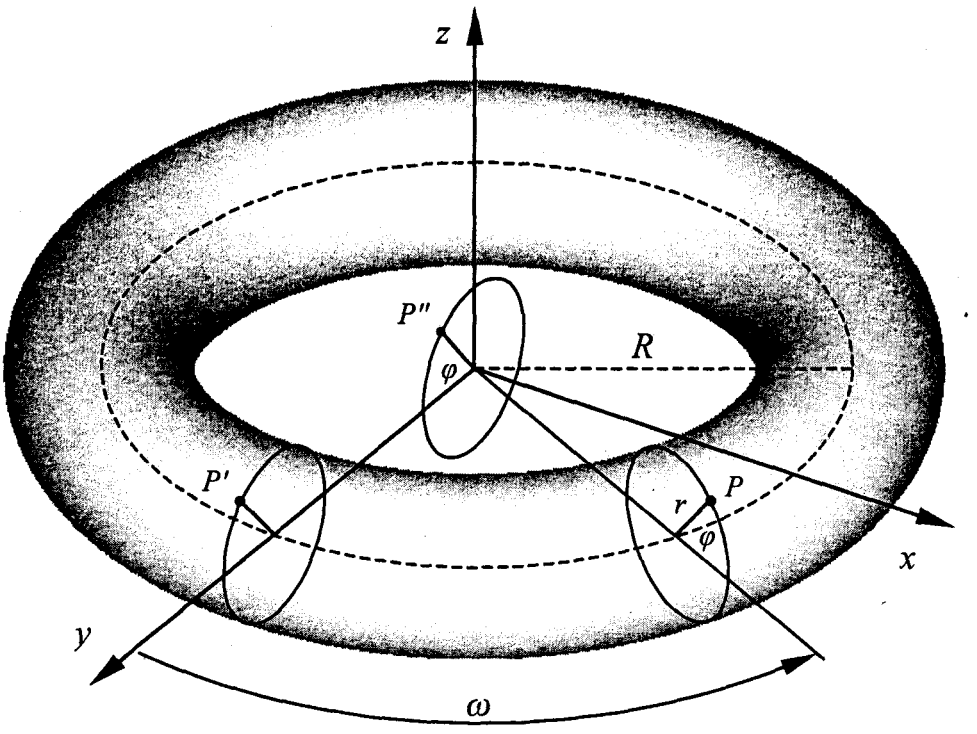


Рис. 2.14. Тор

Непосредственное определение координат точки P представляется сложным, поэтому искомые координаты будем искать несколькими шагами преобразований. Рассмотрим окружность, лежащую в плоскости $z0y$, центр этой окружности совпадает с центром координат. Координаты точки P'' с широтой φ составляют

$$x'' = 0,$$
$$y'' = r\cos\varphi,$$
$$z'' = r\sin\varphi,$$

где r — малый радиус тора.

Теперь перенесем окружность на расстояние R (большой радиус тора) по оси y в той же плоскости $z0y$. Получим точку P'. Ее координаты:

$$x' = x'' = 0,$$
$$y' = R + y'' = R + r\cos\varphi,$$
$$z' = z'' = r\sin\varphi.$$

Окружность, которой принадлежит точка P', является геометрическим местом точек тора с нулевой долготой ω. Если точку P' повернуть на угол ω, то получим искомую точку P поверхности тора с координатами

$$x = x' \cos \omega + y' \sin \omega,$$
$$y = -x' \sin \omega + y' \cos \omega,$$
$$z = z'.$$

Подставляя значения (x', y', z'), получим искомые формулы

$$x = (R + r \cos \varphi) \sin \omega,$$
$$y = (R + r \cos \varphi) \cos \omega,$$
$$z = r \sin \varphi.$$

Эту задачу можно было бы решить, используя преобразование координат. Подобный случай мы рассмотрим ниже (пример studex8 в разделе программирования). Однако, как представляется, более ясным здесь выглядит использование операций перемещения точки (из положения P'' в P', а затем в P).

2.4. Проекции

В настоящее время наиболее распространены устройства отображения, которые синтезируют изображения на плоскости — экране дисплея или бумаге. Устройства, которые создают истинно объемные изображения, пока достаточно редки. Но все чаще появляются сведения о таких разработках, например, об объемных дисплеях [37] или даже о трехмерных принтерах [45].

При использовании любых графических устройств обычно используют проекции. Проекция задает способ отображения объектов на графическом устройстве. Мы будем рассматривать только проекции на плоскость.

Мировые и экранные координаты

При отображении пространственных объектов на экране или на листе бумаги с помощью принтера необходимо знать координаты объектов. Мы рассмотрим две системы координат. Первая — ***мировые координаты***, которые описывают истинное положение объектов в пространстве с заданной точностью. Другая — система координат устройства изображения, в котором осуществляется вывод изображения объектов в заданной проекции.

Пусть мировые координаты будут трехмерными декартовыми координатами. Где должен размещаться центр координат и какими будут единицы измере-

ния вдоль каждой оси, пока для нас не очень важно. Важно то, что для отображения мы будем знать какие-то числовые значения координат отображаемых объектов.

Для получения изображения в определенной проекции необходимо рассчитать координаты проекции. Из них можно получить координаты для графического устройства — назовем их ***экранными координатами***. Для синтеза изображения на плоскости достаточно двумерной системы координат. Однако в некоторых алгоритмах визуализации используются трехмерные экранные координаты, например, в алгоритме *Z*-буфера.

Основные типы проекций

В компьютерной графике наиболее распространены *параллельная* и *центральная* проекции (рис. 2.15).

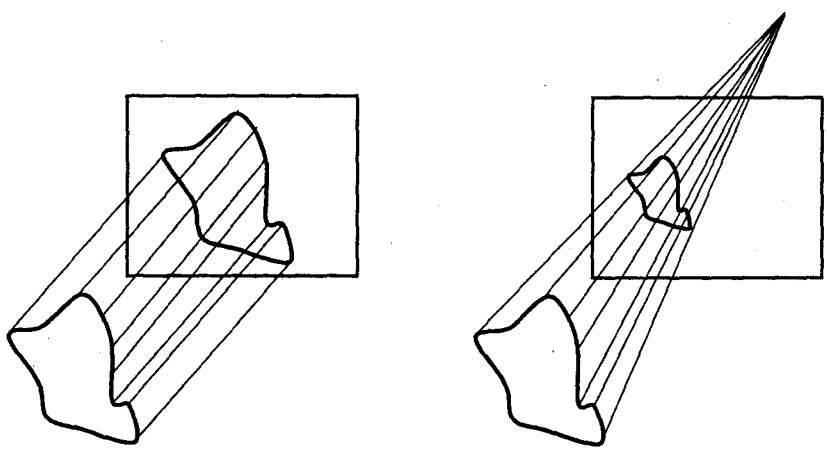


Рис. 2.15. Параллельная и центральная проекции

Для центральной проекции (также называемой *перспективной*) лучи проецирования исходят из одной точки, размещенной на конечном расстоянии от объектов и плоскости проецирования. Для параллельной проекции лучи проецирования параллельны.

Аксонометрическая проекция

Аксонометрическая проекция — разновидность параллельной проекции. Для нее все лучи проецирования располагаются под прямым углом к плоскости проецирования (рис. 2.16).

Зададим положения плоскости проецирования с помощью двух углов — α и β. Расположим камеру так, чтобы проекция оси z на плоскости проецирования $X0Y$ была бы вертикальной линией (параллельной оси $0Y$).

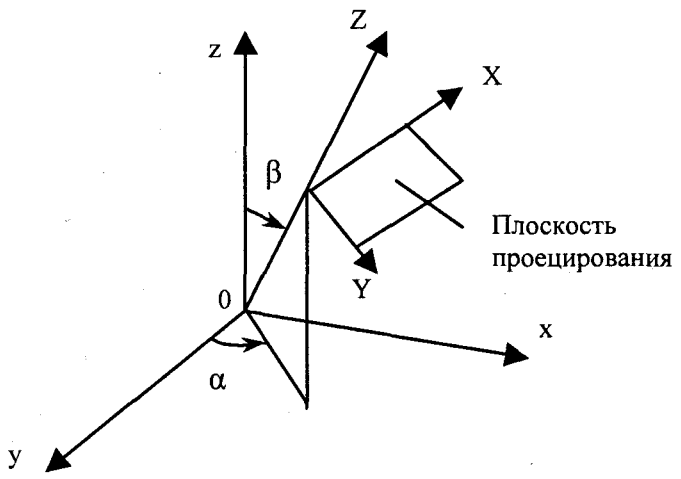


Рис. 2.16. Аксонометрическая проекция

Для того чтобы найти соотношения между координатами (x, y, z) и (X, Y, Z) для любой точки в трехмерном пространстве, рассмотрим преобразования системы координат (x, y, z) в систему (X, Y, Z). Зададим такое преобразование двумя шагами.

1-й шаг. Поворот системы координат относительно оси z на угол α. Такой поворот осей описывается матрицей

$$A = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2-й шаг. Поворачиваем систему координат (x', y', z') относительно оси x' на угол β — получаем координаты (X, Y, Z). Матрица поворота

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta & 0 \\ 0 & \sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Преобразования координат выражаем произведением матриц $B \times A$:

$$B \times A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta & 0 \\ 0 & \sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha\cos\beta & \cos\alpha\cos\beta & -\sin\beta & 0 \\ \sin\alpha\sin\beta & \cos\alpha\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Запишем преобразование для координат проекции в виде формул:

$$X = x\cos\alpha - y\sin\alpha,$$
$$Y = x\sin\alpha\cos\beta + y\cos\alpha\cos\beta - z\sin\beta,$$
$$Z = x\sin\alpha\sin\beta + y\cos\alpha\sin\beta + z\cos\beta.$$

Как вы считаете, будет ли получена та же проекция, если описывать преобразования координат теми же двумя шагами, но в другой последовательности — сначала поворот системы координат относительно оси x на *угол* β, а потом поворот системы координат относительно оси z' на угол α? И будут ли вертикальные линии в системе координат (x,y,z) рисоваться также вертикалями в системе координат (X, Y, Z)? Иначе говоря, выполняется ли $A \times B = B \times A$?

Обратное преобразование координат аксонометрической проекции. Для того, чтобы координаты проекции (X, Y, Z) преобразовать в мировые координаты (x, y, z), нужно проделать обратную последовательность поворотов. Вначале выполнить поворот на угол $-\beta$, а затем — поворот на угол $-\alpha$. Запишем обратное преобразование в матричном виде

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \left[A^{-1} \right] \left[B^{-1} \right] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

Матрицы поворотов:

$$\left[A^{-1} \right] = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

и

$$\left[B^{-1} \right] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & \sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Перемножив матрицы A^{-1} и B^{-1}, получим матрицу обратного преобразования:

$$\left[A^{-1} \right]\left[B^{-1} \right] = \begin{bmatrix} \cos\alpha & \sin\alpha\cos\beta & \sin\alpha\sin\beta & 0 \\ -\sin\alpha & \cos\alpha\cos\beta & \cos\alpha\sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Запишем обратное преобразование также и в виде формул

$$\begin{aligned} x &= X\cos\alpha + Y\sin\alpha\cos\beta + Z\sin\alpha\sin\beta, \\ y &= -X\sin\alpha + Y\cos\alpha\cos\beta + Z\cos\alpha\sin\beta, \\ z &= -Y\sin\beta + Z\cos\beta. \end{aligned}$$

Перспективная проекция

Перспективную проекцию (рис. 2.17) сначала рассмотрим при вертикальном расположении камеры, когда $\alpha = \beta = 0$. Такую проекцию можно себе представить как изображение на стекле, через которое смотрит наблюдатель, расположенный сверху в точке $(x, y, z) = (0, 0, z_k)$. Здесь плоскость проецирования параллельна плоскости $(x\,0\,y)$.

Исходя из подобия треугольников, запишем такие пропорции:

$$\begin{cases} X/(z_k - z_{пл}) = x/(z_k - z), \\ Y/(z_k - z_{пл}) = y/(z_k - z). \end{cases}$$

Учитывая также координату Z:

$$\begin{cases} X = x(z_k - z_{пл})/(z_k - z), \\ Y = y(z_k - z_{пл})/(z_k - z), \\ Z = z - z_{пл}. \end{cases}$$

В матричной форме преобразования координат можно записать так:

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{pmatrix} \frac{z_k - z_{пл}}{z_k - z} & 0 & 0 & 0 \\ 0 & \frac{z_k - z_{пл}}{z_k - z} & 0 & 0 \\ 0 & 0 & 1 & -z_{пл} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

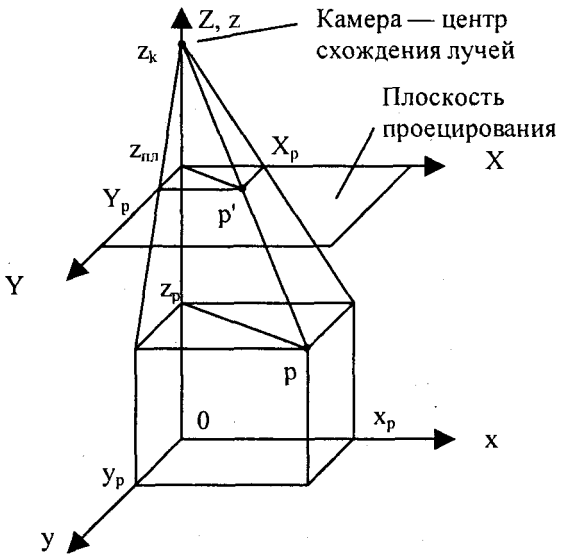


Рис. 2.17. Перспективная проекция

Обратите внимание на то, что здесь коэффициенты матрицы зависят от координаты z (в знаменателе дробей). Это означает, что преобразование координат является нелинейным (а точнее, дробно-линейным), оно относится к классу *проективных* преобразований.

Теперь рассмотрим общий случай — для произвольных углов наклона камеры (α и β) так же, как и для параллельной аксонометрической проекции. Пусть (x', y', z') — координаты для системы координат, повернутой относительно начальной системы (x, y, z) на углы α и β.

Тогда

$$x' = x\cos\alpha - y\sin\alpha,$$
$$y' = x\sin\alpha\cos\beta + y\cos\alpha\cos\beta - z\sin\beta,$$
$$z' = x\sin\alpha\sin\beta + y\cos\alpha\sin\beta + z\cos\beta.$$

Запишем преобразования координат перспективной проекции в виде:

$$X = \Pi x\,(x', z'),$$
$$Y = \Pi y\,(y', z').$$

Последовательность преобразования координат можно описать так :

$$(x, y, z) \xrightarrow{\text{поворот}} (x', y', z') \xrightarrow{\text{перспектива}} (X, Y).$$

Преобразование в целом нелинейное. Его нельзя описать одной матрицей коэффициентов-констант для всех объектов сцены (хотя для преобразования координат можно использовать и матричную форму).

Для такой перспективной проекции плоскость проецирования перпендикулярна лучу, исходящему из центра $(x, y, z) = (0, 0, 0)$ и наклоненному под углом α, β. Если камеру отдалять от центра координат, то центральная проекция видоизменяется. Когда камера в бесконечности, центральная проекция вырождается в параллельную проекцию.

Укажем основные свойства перспективного преобразования. В центральной проекции:

- не сохраняется отношение длин и площадей;
- прямые линии изображаются прямыми линиями;
- параллельные прямые изображаются сходящимися в одной точке.

Последнее свойство широко используется в *начертательной геометрии* для ручного рисования на бумаге. Проиллюстрируем это на примере каркаса домика (рис. 2.18).

Существуют и другие перспективные проекции, которые различаются положением плоскости проецирования и местом точки схождения лучей проецирования. Кроме того, проецирование может осуществляться не на плоскость, а, например, на сферическую или цилиндрическую поверхность.

Рассмотрим косоугольную проекцию, для которой лучи проецирования не перпендикулярны плоскости проецирования. Основная идея такой проек-

ции — камера поднята на высоту h с сохранением вертикального положения плоскости проектирования (рис. 2.19).

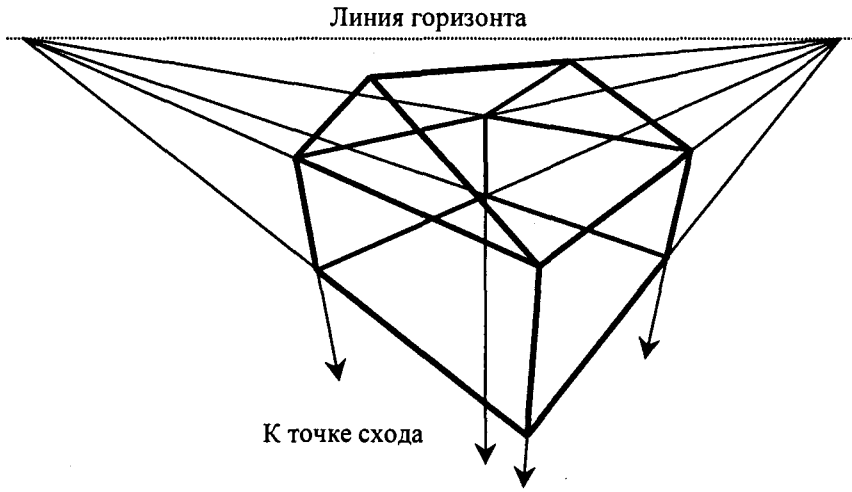


Рис. 2.18. Параллельные линии изображаются в центральной проекции сходящимися в одной точке

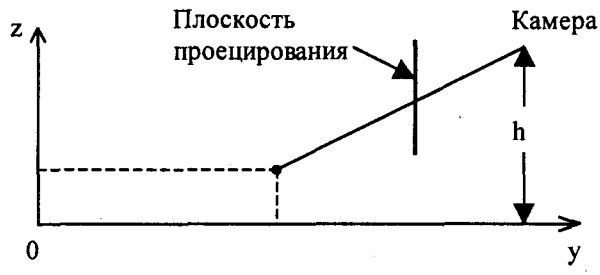


Рис. 2.19. Косоугольная проекция

Получить такую проекцию можно следующим способом:

1. Выполняем поворот вокруг оси z на угол α.
2. Заменяем z' на $-y'$, а y' на z'.
3. Выполняем сдвиг системы координат вверх на высоту камеры h.
4. В плоскости (x', y', 0) строим перспективную проекцию уже рассмотренным выше способом (точка схода лучей на оси z).

Преобразование координат может быть описано таким образом. Сначала определяются (x', y', z').

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ 0 & 0 & -1 & h \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

А потом выполняется перспективное преобразование

$$X = Пx\,(x', z'),$$
$$Y = Пy\,(y', z').$$

Преимущество такой проекции заключается в сохранении параллельности вертикальных линий, что иногда полезно при изображении домов в архитектурных компьютерных системах.

Примеры изображений в различных проекциях. Приведем примеры изображений одинаковых объектов в различных проекциях. В качестве объектов будут кубы одинакового размера. Положение камеры определим углами наклона $\alpha = 27°, \beta = 70°$.

Пример аксонометрической проекции приведен на рис. 2.20.

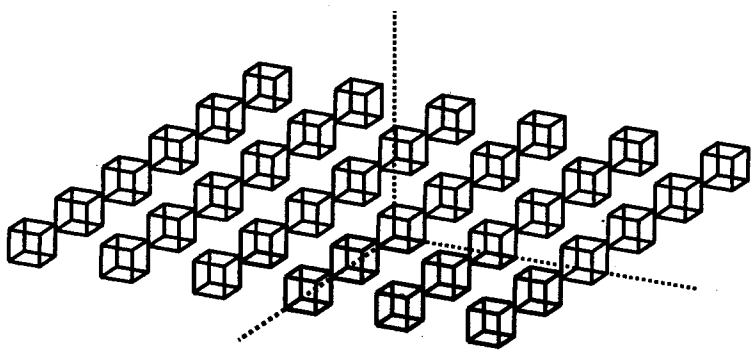


Рис. 2.20. Аксонометрическая проекция

Теперь рассмотрим примеры для перспективной проекции. В отличие от параллельной проекции, изображение в перспективной проекции существенно зависит от положения плоскости проецирования и расстояния до камеры.

В оптических системах известно понятие *фокусного расстояния*. Чем больше фокусное расстояние объектива, тем меньше восприятие перспективы (рис. 2.21), и наоборот, для короткофокусных объективов перспектива наибольшая (рис. 2.22). Данный эффект вы, наверное, уже замечали, если занимались съемками видеокамерой или фотоаппаратом. В наших примерах можно наблюдать некоторое соответствие величины расстояния от камеры до плоскости проецирования ($z_к - z_{пл}$) и фокусного расстояния объектива. Это соответствие, однако, условно, аналогия с оптическими системами здесь неполная.

Для приведенных ниже примеров (рис. 2.21, 2.22) $z_{пл} = 700$. Углы наклона камеры $\alpha = 27°, \beta = 70°$.

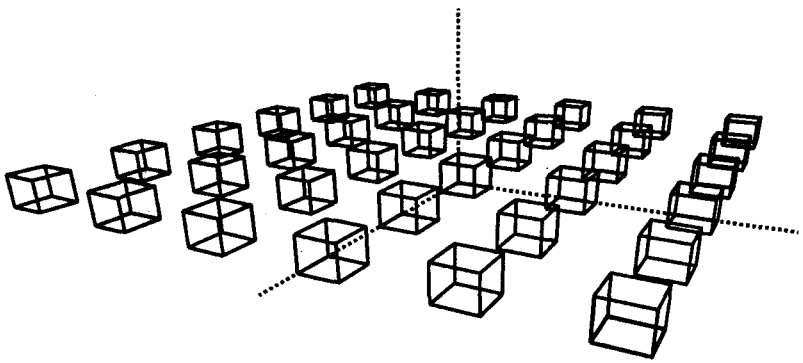


Рис. 2.21. Перспективная проекция для длиннофокусной камеры ($z_к = 2000$)

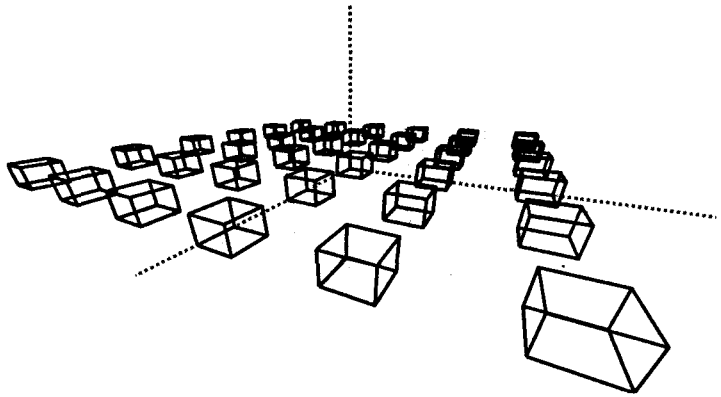


Рис. 2.22. Перспективная проекция для короткофокусной камеры ($z_к = 1200$)

В случае короткофокусной камеры ($z_к$ = 1200) восприятие перспективы наиболее заметно для кубов, которые расположены ближе всего к камере. Вертикальные линии объектов не являются вертикалями на проекции (объекты "разваливаются").

Рассмотрим примеры косоугольной проекции (рис. 2.23, 2.24). Для нее вертикальные линии объектов сохраняют вертикальное расположение на проекции. Положение камеры (точки схождения лучей проецирования) описывается углом поворота α = 27° и высотой подъема h = 500. Плоскость проецирования параллельна плоскости ($x'0y'$) и располагается на расстоянии $z_{пл}$ = 700.

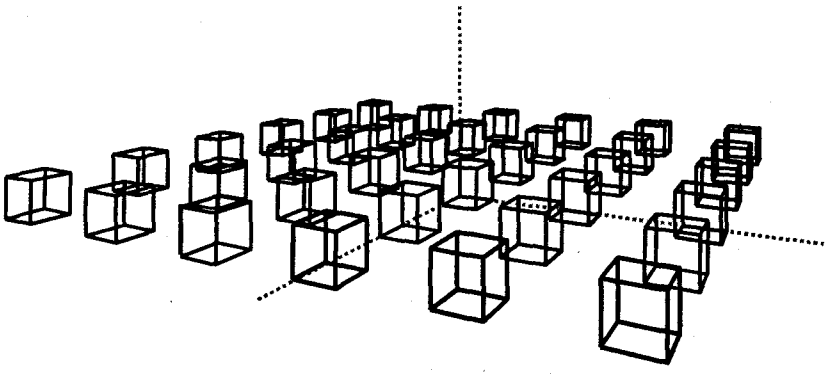


Рис. 2.23. Косоугольная перспективная проекция для длиннофокусной камеры ($z_к$ = 2000)

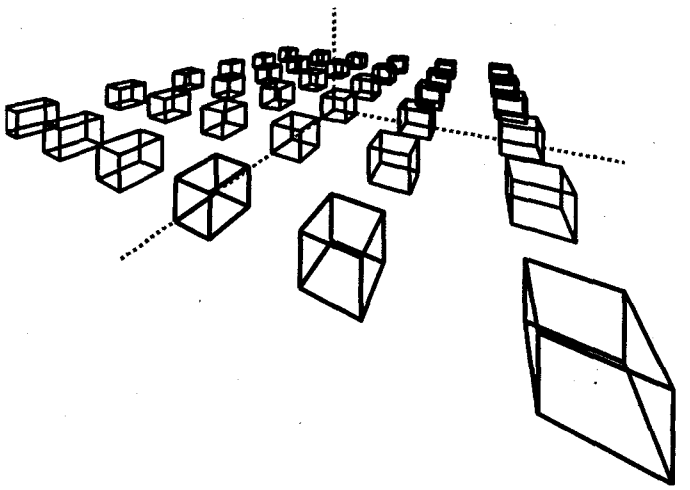
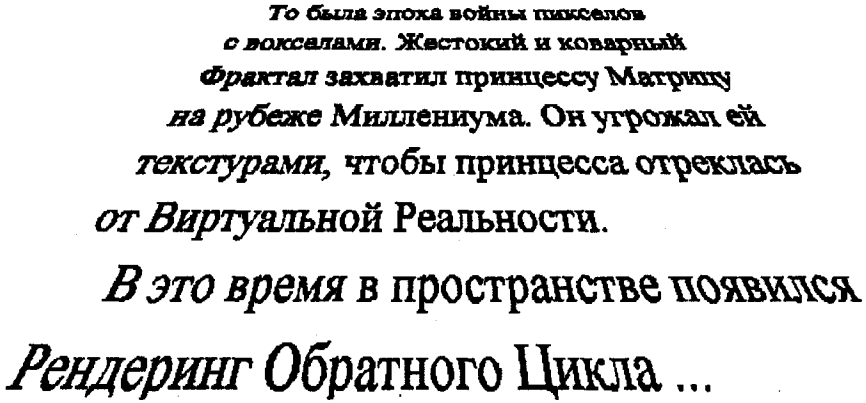


Рис. 2.24. Косоугольная перспективная проекция для короткофокусной камеры ($z_к$ = 1200)

Рассмотрим еще один пример изображения в центральной проекции — текст в стиле фильма "Звездные войны":



Отображение в окне

Как мы уже рассмотрели выше, отображение на плоскость проецирования соответствует некоторому преобразованию координат. Это преобразование координат различно для разных типов проекции, но, так или иначе, осуществляется переход к новой системе координат — координатам проецирования.

Координаты проецирования могут быть использованы для формирования изображения с помощью устройства графического вывода. Однако при этом могут понадобиться дополнительные преобразования, поскольку система координат в плоскости проецирования может не совпадать с системой координат устройства отображения. Например, должны отображаться объекты, измеряемые в километрах, а в растровом дисплее единицей измерения является пиксел. Как выразить километры в пикселах?

Кроме того, вы, наверное, видели, что на экране компьютера можно показывать увеличенное, уменьшенное изображение объектов, а также их перемещать. Как это делается?

Введем обозначения. Пусть ($X_Э$, $Y_Э$, $Z_Э$) — это экранные координаты объектов в графическом устройстве отображения. Заметим, что не следует воспринимать слово "экранные" так, будто речь идет только о дисплеях — все нижеследующее можно отнести и к любым другим устройствам, использующим декартову систему координат. Координаты проецирования обозначим здесь как (X, Y, Z).

Назовем *окном* прямоугольную область вывода с экранными координатами ($X_{Эmin}$ $Y_{Эmin}$) – ($X_{Эmax}$ $Y_{Эmax}$). Обычно приходится отображать в окне или всю сцену, или отдельную ее часть (рис. 2.25).

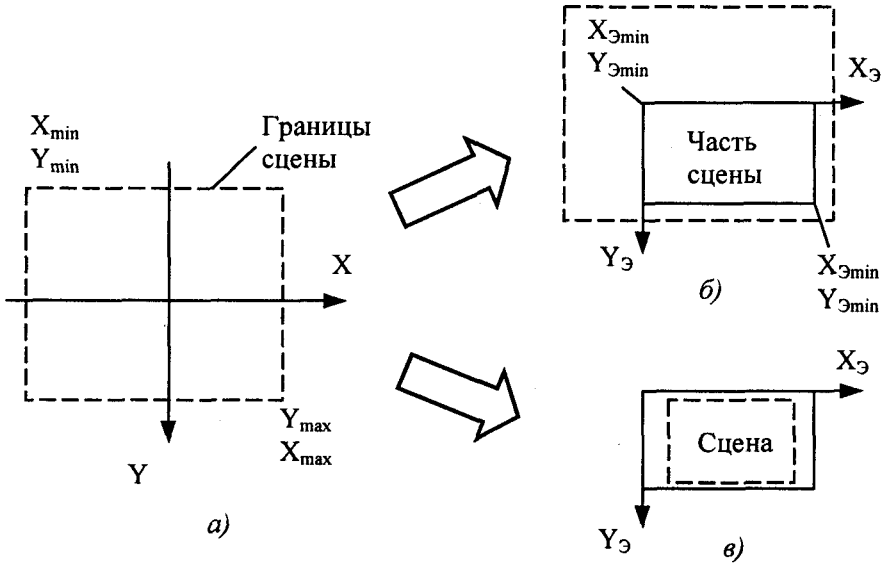


Рис. 2.25. Отображение проекции сцены:
а — границы сцены в координатах проекции; *б* — в окне часть сцены,
в — вся сцена с сохранением пропорций вписана в окно

Преобразование координат проекции в экранные координаты можно задать как растяжение/сжатие и сдвиг:

$$\begin{cases} X_Э = KX + dx, \\ Y_Э = KY + dy, \\ Z_Э = KZ. \end{cases}$$

Такое преобразование сохраняет пропорции объектов благодаря одинаковому коэффициенту растяжения/сжатия (K) для всех координат. Заметим, что для плоского отображения можно отбросить координату Z.

Рассмотрим, как можно вычислить K, dx и dy. Например, необходимо вписать все изображение сцены в окно заданных размеров. Условие вписывания можно определить так:

$$\begin{cases} X_{\text{Э min}} \le KX_{\min} + dx, & (1) \\ Y_{\text{Э min}} \le KY_{\min} + dy, & (2) \\ X_{\text{Э max}} \ge KX_{\max} + dx, & (3) \\ Y_{\text{Э max}} \ge KY_{\max} + dy. & (4) \end{cases}$$

Если прибавить (1) к (3), то получим:

$$K \le \frac{X_{\text{Эmax}} - X_{\text{Эmin}}}{X_{\max} - X_{\min}} = K_X.$$

Из неравенств (2) и (4) следует:

$$K \le \frac{Y_{\text{Эmax}} - Y_{\text{Эmin}}}{Y_{\max} - Y_{\min}} = K_Y.$$

Решением системы (1)—(4) для K будет: $K \le \min\{K_X, K_Y\} = K_{\min}$.

Если значение K_X или значение K_Y равно бесконечности, то его необходимо отбросить. Если оба — то значение $K_{\min}$ можно задать равным единице. Для того чтобы изображение в окне имело наибольший размер, выберем $K = K_{\min}$. Теперь можно найти dx. Из неравенства (1):

$$dx \ge X_{\text{Эmin}} - KX_{\min} = dx_1.$$

Из неравенства (3):

$$dx \le X_{\text{Эmax}} - KX_{\max} = dx_2.$$

Поскольку $dx_1 \le dx_2$, то величину dx можно выбрать из интервала $dx_1 \le dx \le dx_2$. Выберем центральное расположение в окне:

$$dx = \frac{dx_1 + dx_2}{2} = \frac{X_{\text{Эmin}} - KX_{\min} + X_{\text{Эmax}} - KX_{\max}}{2}.$$

Аналогично найдем dy:

$$dy = \frac{Y_{\text{Эmin}} - KY_{\min} + Y_{\text{Эmax}} - KY_{\max}}{2}.$$

При таких значениях dx и dy центр сцены будет в центре окна.

В других случаях, когда в окне необходимо показывать с соответствующим масштабом лишь часть сцены, можно прямо задавать числовые значения масштаба (K) и координаты сдвига (dx, dy). При проектировании интерфейса графической системы желательно ограничить выбор K, dx, dy диапазоном допустимых значений.

В графических системах используются разнообразные способы задания масштаба отображения и определения границ сцены для показа в окне просмотра. Например, для сдвига часто используют ползунки скроллинга. Также можно указывать курсором точку на сцене, и затем эта точка становится центральной точкой окна. Или можно очертить прямоугольник, выделяя границы фрагмента сцены, — тогда этот фрагмент затем будет вписан в окно. И так далее. Все эти способы отображения основываются на растяжении/сжатии (масштабировании), а также сдвиге, и описываются аффинным преобразованием координат.

2.5. Выводы

Представим цепочку преобразований координат от мировых к экранным следующим образом (рис. 2.26):

Рис. 2.26. Этапы преобразований координат

Для аксонометрической (параллельной) проекции координаты проекции совпадают с видовыми координатами (хотя это и не обязательно). Преобразование координат можно описать одной матрицей, которая получается перемножением соответствующих матриц.

$$\begin{bmatrix} X_Э \\ Y_Э \\ Z_Э \\ 1 \end{bmatrix} = \begin{bmatrix} K,\ dx,\ dy \end{bmatrix} \begin{bmatrix} \text{Повороты} \\ \alpha,\ \beta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} M_A \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

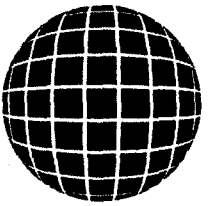
Важно то, что для аксонометрической проекции коэффициенты матрицы — это константы, одинаковые для всех точек трехмерного пространства. Это дает возможность свести к минимуму вычисления координат в цикле графического вывода. В этом плане можно отметить крайний случай, когда мировые координаты совпадают с экранными — вообще нет никаких преобразований координат. Например, координаты объектов задаются в пикселах экрана. Такое часто встречается, например, в двухмерной графике.

Для перспективной (центральной) проекции коэффициенты матрицы проецирования на плоскость не являются константами — они зависят от Z. Это делает нецелесообразным запись цепочки преобразований в виде одной матрицы и, как следствие этого, усложняется расчет координат в сравнении с параллельной проекцией.

$$\begin{bmatrix} X_Э \\ Y_Э \\ Z_Э \\ 1 \end{bmatrix} = \begin{bmatrix} K,\ dx,\ dy \end{bmatrix} \begin{bmatrix} \text{Проецирование} \end{bmatrix} \begin{bmatrix} \text{Повороты} \\ \alpha,\ \beta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Для центральной проекции иногда используют матричную форму с применением обобщенных однородных нелинейных координат [28, 32].

В качестве мировых и экранных координат нами была использована трехмерная ортогональная система. В компьютерных графических системах также используются другие системы координат и иные проекции. В особенности это касается систем, которые моделируют объекты, располагающиеся на поверхности Земли. С этими вопросами можно ознакомиться в многочисленной литературе по геодезии и картографии, а также в работах, посвященных *геоинформационным* системам.



ГЛАВА 3

Базовые растровые алгоритмы

Сформировать растровое изображение можно по-разному. Для того чтобы создать изображение на растровом дисплее, можно просто скопировать готовый растр в видеопамять. Этот растр может быть получен, например, с помощью сканера или цифрового фотоаппарата. А можно создавать изображение объекта путем последовательного рисования отдельных простых элементов.

Простые элементы, из которых складываются сложные объекты, будем называть ***графическими примитивами***. Их можно встретить повсюду. Например, для построения изображения используется некоторый набор примитивов, которые поддерживаются определенными графическими устройствами вывода. Графические примитивы также можно применять для описания пространственных объектов в базе данных компьютерной системы — модели объектов. Могут использоваться различные множества примитивов для модели и для алгоритмов отображения. Удобно, когда эти множества совпадают, тогда упрощается процесс отображения.

Простейшим и, вместе с тем, наиболее универсальным растровым графическим примитивом является пиксел. Любое растровое изображение можно нарисовать по пикселам, но это сложно и долго. Необходимы больше сложные элементы, для которых рисуются сразу несколько пикселов.

Рассмотрим графические примитивы, которые используются наиболее часто в современных графических системах, — это линии и фигуры.

3.1. Алгоритмы вывода прямой линии

Рассмотрим растровые алгоритмы для отрезков прямой линии. Предположим, что заданы координаты ($x1,y1 - x2,y2$) концов отрезка прямой. Для вы-

вода линии необходимо закрасить в определенный цвет все пикселы вдоль линии. Для того чтобы закрасить каждый пиксел, необходимо знать его координаты.

Наиболее просто нарисовать отрезок горизонтальной линии:

```
for (x=x1; x<=x2; x++)
    Пиксел(x, y1);
```

Вычисление текущих координат пиксела здесь выполняется как приращение по x (необходимо, чтобы $x1<=x2$), а вывод пиксела обеспечивается функцией Пиксел(). Поскольку в языке С, С++ для названия функции нельзя использовать кириллицу, то будем дальше использовать ее как комментарий.

Аналогично рисуется отрезок вертикали:

```
for (y=y1; y<=y2; y++)
   //Пиксел(x1, y);
```

Как видим, в цикле выполняются простейшие операции над целыми числами — приращение на единицу и проверка на "< =". Поэтому операция рисования отрезка выполняется быстро и просто. Ее используют как базовую операцию для других операций, например, в алгоритмах заполнения плоскости полигонов.

Можно поставить такой вопрос: какая линия рисуется быстрее — горизонталь или вертикаль? На первый взгляд — одинаково быстро. Если учитывать только математические аспекты, то скорость должна быть одной и той же при одинаковой длине линий, поскольку в обоих случаях выполняется равное количество идентичных операций. Однако если кроме расчета координат анализировать также вывод пикселов, то могут быть отличия. В растровых системах рисование пиксела обычно означает запись одного или нескольких бит в память, где сохраняется растр. И здесь уже не все равно — по строкам или по столбцам заполняется растр. Необходимо учитывать логическую организацию памяти компьютера, в которой хранятся биты или байты растра. Даже для компьютеров одного типа (например, персональных компьютеров) для различных поколений процессоров и памяти скорость записи по соседним адресам может существенно отличаться от скорости записи по не соседним адресам [40]. В особенности это заметно, когда для растра используется виртуальная память с сохранением отдельных страниц на диске и (или) в оперативной памяти (RAM). При работе графических программ в среде операционной системы Windows часто случается так, что горизонтали рисуются быстрее вертикалей, поскольку в страницах памяти хранятся соседние байты. А может быть, что RAM достаточно, но скорости рисования все же различны.

Например, если используется черно-белый растр в формате один бит на пиксел, то для вертикали битовая маска одинакова для всех пикселов линии, а для горизонтали маску нужно изменять на каждом шаге. Здесь необходимо заметить, что рисование черно-белых горизонталей можно существенно ускорить, если записывать сразу восемь соседних пикселов — байт в памяти.

Горизонтали и вертикали представляют собой частный случай линий. Рассмотрим линию общего вида. Для нее также необходимо вычислять координаты каждого пиксела. Известно несколько методов расчетов координат точек линии.

Прямое вычисление координат

Пусть заданы координаты конечных точек отрезка прямой. Найдем координаты точки внутри отрезка (рис. 3.1).

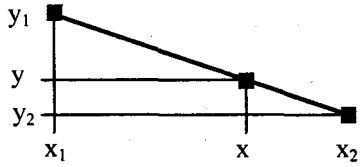


Рис. 3.1. Отрезок прямой

Запишем отношения катетов для подобных треугольников:

$$\frac{x-x_1}{y-y_1}=\frac{x_2-x_1}{y_2-y_1}.$$

Тогда

$$x=x_1+(y-y_1)\frac{x_2-x_1}{y_2-y_1},$$

то есть $x=f(y)$.

А также

$$y=y_1+(x-x_1)\frac{y_2-y_1}{x_2-x_1},$$

то есть $y=F(x)$.

В зависимости от угла наклона прямой выполняется цикл по оси x или по y (рис. 3.2).

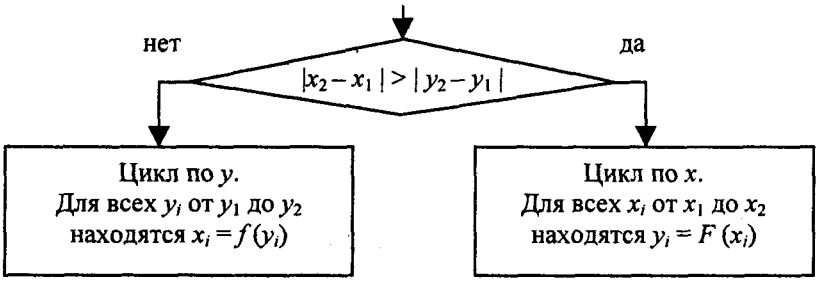


Рис. 3.2. Алгоритм вывода отрезка прямой линии

Приведем пример записи этого алгоритма на компьютерном языке программирования С, С++. Для сокращения текста рассмотрим фрагмент программы, где выполняется цикл по оси x, причем $x1<x2$:

```
for (x=x1; x<=x2; x++)
  {
  y=y1+((x-x1)*(y2-y1))/(x2-x1);
  // Пиксел(x,y);
  }
```

Здесь все операции выполняются над целыми числами. Двойные скобки необходимы для того, чтобы деление выполнялось после умножения. Недостатки такой программы — в цикле выполняется много лишних операций, присутствуют операции деления и умножения. Это обуславливает малую скорость работы. Относительно лишних операций в цикле. Можно вынести вычисление $(y2-y1)/(x2-x1)$ из цикла, поскольку это значение не изменяется. Однако для этого уже необходимо использовать операции с плавающей точкой:

```
float k;

k=(float)(y2-y1)/(float)(x2-x1);
for (x=x1; x<=x2; x++)
  {
  y=y1+(float)(x-x1)*k;
  //Пиксел(x,y);
  }
```

Поскольку мы решили использовать операции с плавающей точкой, то попробуем еще уменьшить количество операций в цикле. Если раскрыть скобки в выражении $y = y1 + (x - x1)\cdot k$; то получим $y = y1 + x\cdot k - x1\cdot k$. Здесь значение $(y1 - x1\cdot k)$ является константой — эти операции также вынесем из тела цикла.

```
float yy,k;
k =(float)(y2-y1)/(float)(x2-x1);
yy=(float)y1 - (float)x1*k;
for (x=x1; x<=x2; x++)
  {
  y = yy + (float)x*k;
  //Пиксел(x,y);
  }
```

В цикле выполняются только две арифметические операции и преобразование x из целого в форму с плавающей точкой.

Если рассматривать цикл вычисления y_i по соответствующим значениям $x_i = x1, x1+1, \ldots, x2$ как итеративный процесс, то можно поставить такой вопрос: чему равна разность $(y_{i+1} - y_i)$? Она равна $y_{i+1} - y_i = x1 + (x_{i+1} - x1)\,k - x1 - (x_i - x1)\,k = (x_{i+1} - x_i)\,k = k$; поскольку $x_{i+1} - x_i = 1$. Разность $(y_{i+1} - y_i)$ — константа, равная k. Исходя из этого, можно построить цикл таким образом:

```
float k;
k = (float)(y2-y1)/(float)(x2-x1);
y = y1;
for (x=x1; x<=x2; x++)
  {
  //Пиксел (x,y);
  y += k;
  }
```

В теле цикла есть только одна операция для вычисления координаты y (если не учитывать <= и ++).

Если сравнивать последний вариант с предыдущим, то последний лучше по быстродействию. Также существенно отличаются способы вычисления координаты y. В последнем варианте значение y вычисляется прибавлением приращения k на каждом шаге, и на последнем шаге цикла (когда x=$x2$) должно стать y=$y2$. Исходя из чисто математических соображений здесь все корректно, однако необходимо учесть, что в компьютере дробные числа представляются в формате с плавающей точкой не точно. Кроме погрешности представления чисел существует ошибка выполнения арифметических операций с плавающей точкой. Ошибка зависит от разрядности мантисс, и самая малая — для **long double**, но все равно не нулевая. С каждым шагом цикла ошибки накапливаются, и может так произойти, что y не равно $y2$ на последнем шаге. Это необходимо учитывать при использовании алгоритма.

Положительные черты прямого вычисления координат:

1. Простота, ясность построения алгоритма.
2. Возможность работы с нецелыми значениями координат отрезка. (Как вы считаете, какой вариант из четырех корректно вычисляет координаты пикселов, если $x1$, $y1$, $x2$ и $y2$ — дробные?)

Недостатки:

1. Использование операций с плавающей точкой или целочисленных операций умножения и деления обуславливает малую скорость. Однако это зависит от процессора и для различных типов компьютеров может быть поразному. В современных компьютерах, в которых процессоры используют эффективные способы ускорения (например, конвейер арифметических операций с плавающей точкой), время выполнения целочисленных операций уже не намного меньше. Для старых компьютеров разница могла быть в десятки раз, поэтому и старались разрабатывать алгоритмы только на основе целочисленных операций.
2. При вычислении координат путем добавления приращений может накапливаться ошибка вычислений координат.

Последнюю разновидность прямого вычисления координат, рассмотренную здесь, можно было бы назвать "инкрементной". Но такое название получили алгоритмы другого типа.

Инкрементные алгоритмы

Брезенхэм предложил подход [54, 55], позволяющий разрабатывать так называемые *инкрементные* алгоритмы растеризации. Основной целью для разработки таких алгоритмов было построение циклов вычисления координат на основе только целочисленных операций сложения/вычитания без использования умножения и деления. Уже известны инкрементные алгоритмы не только для отрезков прямых, но и для кривых линий [33, 55].

Инкрементные алгоритмы выполняются как последовательное вычисление координат соседних пикселов путем добавления приращений координат. Приращения рассчитываются на основе анализа функции погрешности. В цикле выполняются только целочисленные операции сравнения и сложения/вычитания. Достигается повышение быстродействия для вычислений каждого пиксела по сравнению с прямым способом. Один из вариантов алгоритма Брезенхэма:

```
xerr = 0, yerr =0;
dx = x2 - x1, dy = y2 - y1;
```

```
Если dx > 0, то incX = 1;
     dx = 0, то incX = 0;
     dx < 0, то incX = -1;
Если dy > 0, то incY = 1;
     dy = 0, то incY = 0;
     dy < 0, то incY = -1;
dx = |dx|, dy = |dy|;
Если dx > dy, то d = dx;
              иначе d = dy;
x = x1, y = y1;
Пиксел (x, y)
Выполнить цикл d раз:
   {
   xerr = xerr + dx;
   yerr = yerr + dy;
   Если xerr > d, то xerr = xerr - d
                     x = x + incX
   Если yerr > d, то yerr = yerr - d
                     y = y + incY
   Пиксел (x, y)
   }
```

Рассмотрим пример работы приведенного выше алгоритма Брезенхэма для отрезка $(x1y1 - x2y2) = (2,3 - 8,6)$. Этот алгоритм восьмисвязный, то есть при вычислении приращений координат для перехода к соседнему пикселу возможны восемь случаев (рис. 3.3).

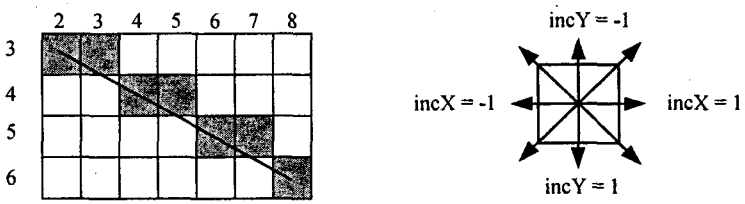


Рис. 3.3. Восьмисвязность

Известны также четырехсвязные алгоритмы (рис. 3.4).

Четырехсвязные алгоритмы проще, но они генерируют менее качественное изображение линий за большее количество тактов роботы. Для приведенного примера четырехсвязный алгоритм работает 10 тактов, а восьмисвязный — только 7.

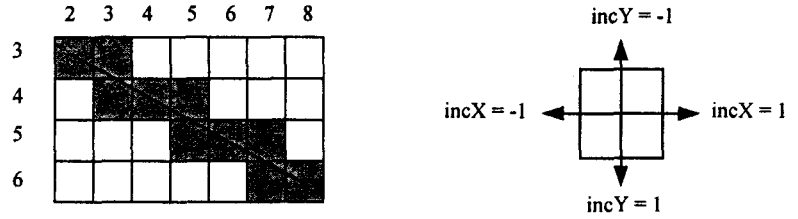


Рис. 3.4. Четырехсвязность

3.2. Алгоритм вывода окружности

Для вывода контура круга можно использовать соотношение между координатами X и Y для точек круга $X^2 + Y^2 = R^2$ и построить алгоритм прямого вычисления координат. Однако тогда необходимо вычислять квадратный корень, а это в цифровом компьютере выполняется медленно.

Дадим запись инкрементного алгоритма Брезенхэма на языке C++:

```
void Circle(int xc,int yc,int radius)
{
int x,y,dxt;
long r2,dst,t,s,e,ca,cd,indx;

r2 = (long)radius*(long)radius;
dst = 4*r2;
dxt = (double)radius/1.414213562373;
t = 0;
s = -4*r2*(long)radius;
e = (-s/2) - 3*r2;
ca = -6*r2;
cd = - 10*r2;
x = 0;
y = radius;
//ПИКСЕЛ (xc,yc+radius);
//ПИКСЕЛ (xc,yc-radius);
//ПИКСЕЛ (xc+radius,yc);
//ПИКСЕЛ (xc-radius,yc);
for (indx=1; indx<=dxt; indx++)
    {
    x++;
    if (e >= 0) e += t + ca;
```

```
    else
       {
       y--;
       e += t - s + cd;
       s += dst;
       }
    t-= dst;
    //ПИКСЕЛ (xc+x,yc+y);
    //ПИКСЕЛ (xc+y,yc+x);
    //ПИКСЕЛ (xc+y,yc-x);
    //ПИКСЕЛ (xc+x,yc-y);
    //ПИКСЕЛ (xc-x,yc-y);
    //ПИКСЕЛ (xc-y,yc-x);
    //ПИКСЕЛ (xc-y,yc+x);
    //ПИКСЕЛ (xc-x,yc+y);
    }
}
```

В этом алгоритме использована симметрия круга — в основном цикле вычисляются координаты точек круга только для одного октанта и сразу рисуются восемь симметрично расположенных пикселов [33].

3.3. Алгоритм вывода эллипса

Инкрементный алгоритм для эллипса подобен алгоритму для круга, но более сложный. Этот алгоритм приведен в [33].

```
void Ellipse(int xc,int yc,int enx,int eny)
{
int x,y;
long a2,b2,dds,ddt,dxt,t,s,e,ca,cd,indx;
int a,b;
a = abs(enx - xc); b = abs(eny - yc);
a2 = (long)a*(long)a;
b2 = (long)b*(long)b;
dds = 4*a2;
ddt = 4*b2;
dxt = (float)a2/sqrt(a2+b2);
t = 0;
s = -4*a2*b;
e = (-s/2) - 2*b2 - a2;
ca = -6*b2;
cd = ca - 4*a2;
x = xc;
y = yc + b;
```

```
//ПИКСЕЛ (x,y);
//ПИКСЕЛ (x,2*yc-y);
//ПИКСЕЛ (2*xc-x,2*yc-y);
//ПИКСЕЛ (2*xc-x,y);
for (indx=1; indx<=dxt; indx++)
    {
    x++;
    if (e >= 0) e += t + ca;
    else
       {
       y--;
       e += t - s +cd;
       s += dds;
       }
    t-= ddt;
    //ПИКСЕЛ (x,y);
    //ПИКСЕЛ (x,2*yc-y);
    //ПИКСЕЛ (2*xc-x,2*yc-y);
    //ПИКСЕЛ (2*xc-x,y);
    }
dxt = abs(y - yc);
e-= t/2 + s/2 + b2 + a2;
ca = -6*a2;
cd = ca - 4*b2;
for (indx=1; indx<=dxt; indx++)
    {
    y--;
    if (e <= 0) e+= -s + ca;
    else
       {
       x++;
       e+= - s + t + cd; t-= ddt;
       }
    s+= dds;
    //ПИКСЕЛ (x,y);
    //ПИКСЕЛ (x,2*yc-y);
    //ПИКСЕЛ (2*xc-x,2*yc-y);
    //ПИКСЕЛ (2*xc-x,y);
    }
}
```

В этом алгоритме использована симметрия эллипса по квадрантам (рис. 3.5). Алгоритм состоит из двух циклов. Сначала от $x=0$ до $x=dxt$, где

$$dxt = \frac{a^2}{\sqrt{a^2 + b^2}},$$

а потом цикл до точки $x = a,\ \ y = 0$.

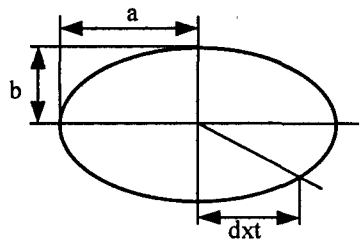


Рис. 3.5. Вывод эллипса

3.4. Кривая Безье

Разработана математиком *Пьером Безье*. Кривые и поверхности Безье были использованы в 60-х годах компанией "Рено" для компьютерного проектирования формы кузовов автомобилей. В настоящее время они широко используются в компьютерной графике.

Кривые Безье описываются в параметрической форме:

$$x = P_x(t),$$
$$y = P_y(t).$$

Значение t выступает как параметр, которому отвечают координаты отдельной точки линии. Параметрическая форма описания может быть более удобной для некоторых кривых, чем задание в виде функции $y=f(x)$. Это потому, что функция $f(x)$ может быть намного сложнее, чем $P_x(t)$ и $P_y(t)$, кроме того, $f(x)$ может быть неоднозначной.

Многочлены Безье для P_x и P_y имеют такой вид:

$$P_x(t) = \sum_{i=0}^{m} C_m^i \, t^i (1-t)^{m-i} x_i,$$

$$P_y(t) = \sum_{i=0}^{m} C_m^i \, t^i (1-t)^{m-i} y_i,$$

где C_m^i — сочетание m по i (известное также по биному Ньютона), $C_m^i = m! /(i! \, (m-i)!)$, а x_i и y_i — координаты точек-ориентиров P_i. Значение m можно рассматривать и как степень полинома, и как значение, которое на единицу меньше количества точек-ориентиров.

Рассмотрим кривые Безье, классифицируя их по значениям m.

$m = 1$ (по двум точкам).

Кривая вырождается в отрезок прямой линии, определяемый концевыми точками P_0 и P_1, как показано на рис. 3.6.

$$P(t) = (1 - t)\,P_0 + t\,P_1\,.$$

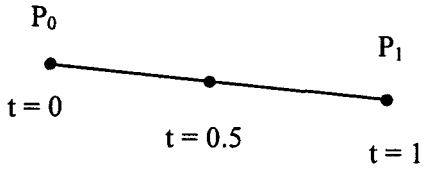


Рис. 3.6. Кривая Безье (m = 1)

$m = 2$ (по трем точкам, рис. 3.7)

$$P(t) = (1 - t)^2\,P_0 + 2t(1 - t)\,P_1 + t^2\,P_2.$$

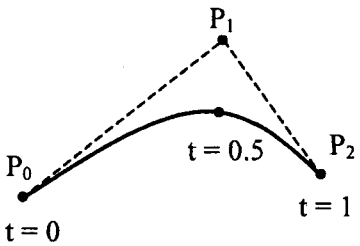


Рис. 3.7. Кривая Безье (m=2)

m = 3 (по четырем точкам, кубическая, рис. 3.8). Используется довольно часто, в особенности в сплайновых кривых.

$$P(t) = (1 - t)^3\,P_0 + 3t(1 - t)^2\,P_1 + 3t\,(1 - t)^2\,P_2 + t^3\,P_3.$$

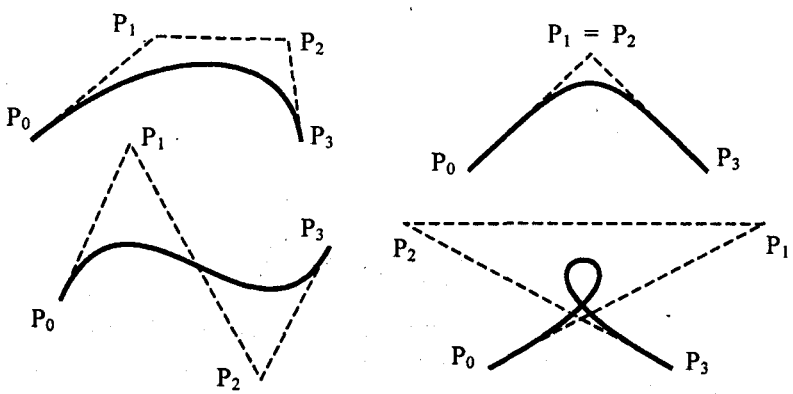


Рис. 3.8. Кубические кривые Безье (m = 3)

Геометрический алгоритм для кривой Безье

Этот алгоритм позволяет вычислить координаты (x, y) точки кривой Безье по значению параметра t. Алгоритм описан в [19].

1. Каждая сторона контура многоугольника, проходящего по точкам-ориентирам, делится пропорционально значению t.
2. Точки деления соединяются отрезками прямых и образуют новый многоугольник. Количество узлов нового контура на единицу меньше, чем количество узлов предыдущего контура.
3. Стороны нового контура снова делятся пропорционально значению t. И так далее. Это продолжается до тех пор, пока не будет получена единственная точка деления. Эта точка и будет точкой кривой Безье (рис. 3.9).

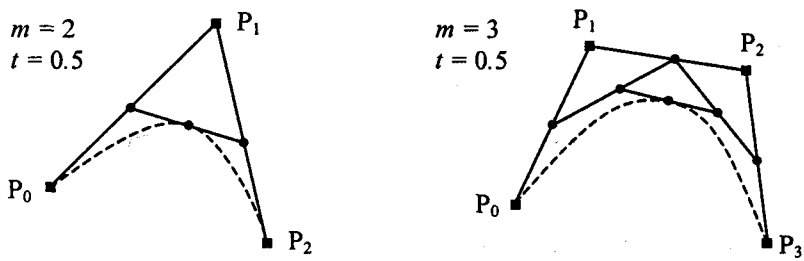


Рис. 3.9. Геометрический алгоритм для кривых Безье

Приведем запись геометрического алгоритма на языке C++:

```
for ( i = 0; i <= m; i++)
    R[i] = P[i]; //формируем вспомогательный массив R[ ]
for ( j = m; j > 0; j--)
  for ( i = 0; i < j; i++)
      R[i] = R[i] + t * (R[i+1] - R[i]);
```

Результат работы алгоритма — координаты одной точки кривой Безье — записываются в R[0].

3.5. Алгоритмы вывода фигур

Фигурой здесь будем считать плоский геометрический объект, который состоит из линий контура и точек, которые содержатся внутри контура.

В общем случае линий контура может быть несколько — когда объект имеет внутри пустоты. В этом случае для описания таких фигур необходимы два и более контуров (рис. 3.10).

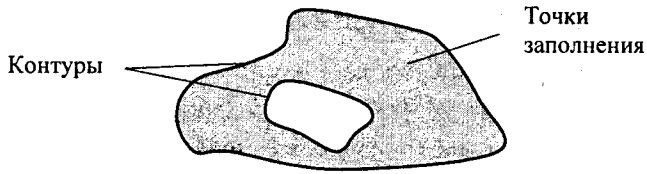


Рис. 3.10. Пример фигуры

В некоторых графических системах одним объектом может считаться и более сложная многоконтурная фигура — совокупность островов с пустотами.

Графический вывод фигур разделяется на две задачи: вывод контура и вывод точек заполнения. Поскольку контур представляет собой линию, то вывод контура проводится на основе алгоритмов вывода линий. В зависимости от сложности контура, это могут быть отрезки прямых, кривых или произвольная последовательность соседних пикселов.

Для вывода точек заполнения известны методы, разделяющиеся в зависимости от использования контура на два типа — алгоритмы закрашивания от внутренней точки к границам произвольного контура и алгоритмы, которые используют математическое описание контура.

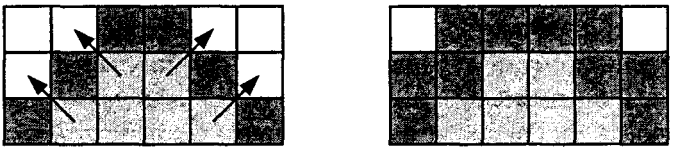
Алгоритмы закрашивания

Рассмотрим алгоритмы закрашивания произвольного контура, который уже нарисован в растре. Сначала находится пиксел внутри контура фигуры. Цвет этого пиксела изменяем на нужный цвет заполнения. Потом проводится анализ цветов всех соседних пикселов. Если цвет некоторого соседнего пиксела не равен цвету границы контура или цвету заполнения, то цвет этого пиксела изменяется на цвет заполнения. Потом анализируется цвет пикселов, соседних с предыдущими. И так далее, до тех пор, пока внутри контура все пикселы не перекрасятся в цвет заполнения.

Пикселы контура — это граница, за которую нельзя выходить в ходе последовательного перебора всех соседних пикселов. Соседними могут считаться только четыре пиксела (справа, слева, сверху и снизу — четырехсвязность), или все восемь пикселов (восьмисвязность). Не всякий контур может служить границей закрашивания (рис. 3.11).

Простейший алгоритм закрашивания. Для всех алгоритмов закрашивания нужно задавать начальную точку внутри контура с координатами x_0, y_0. Простейший алгоритм можно описать так [32]:

```
Шаг 1. Определить x0, y0.
Шаг 2. Выполнить функцию ЗАКРАШИВАНИЕ(x0, y0).
```



Выход за границу контура на следующих шагах работы

Для этого контура выхода за границу не будет

Рис. 3.11. Особенности восьмисвязного закрашивания

Функцию ЗАКРАШИВАНИЕ() определим так:

```
Функция ЗАКРАШИВАНИЕ(x, y)
{
Если цвет пиксела (x, y) не равен цвету границы, то
   {
   установить для пиксела (x, y) цвет заполнения;
   ЗАКРАШИВАНИЕ(x+1, y);
   ЗАКРАШИВАНИЕ(x-1, y);
   ЗАКРАШИВАНИЕ(x, y+1);
   ЗАКРАШИВАНИЕ(x, y-1);
   }
}
```

Такое определение функции является рекурсивным. Рекурсия позволяет упростить запись некоторых алгоритмов. Но для этого алгоритма рекурсия порождает существенные проблемы — рекурсивные вызовы функции ЗАКРАШИВАНИЕ() делаются для каждого пиксела, что обычно приводит к переполнению стека в ходе выполнения компьютерной программы. Практика показывает, что этот алгоритм неприемлем для фигур площадью в тысячу и больше пикселов.

Можно построить подобный алгоритм и без рекурсии, если вместо стека компьютера использовать отдельные массивы. Тогда стек не переполняется.

Волновой алгоритм закрашивания. Алгоритм был разработан авторами работы [38] и предназначался для расчета центра тяжести объектов по соответствующим изображениям. Идея была навеяна волновым алгоритмом поиска трассы на графе, известным в САПР электронных схем. Суть подобных алгоритмов состоит в том, что для начальной точки (вершины на графе) находятся соседние точки (другие вершины), которые отвечают двум условиям: во-первых — эти вершины связаны с начальной; во-вторых — эти вершины еще не отмечены, то есть они рассматриваются впервые. Соседние вершины

текущей итерации отмечаются в массиве описания вершин, и каждая из них становится текущей точкой для поиска новых соседних вершин в следующей итерации. Если в специальном массиве отмечать каждую вершину номером итерации, то когда будет достигнута конечная точка, можно совершить обратный цикл — от конечной точки к начальной по убыванию номеров итераций. В ходе обратного цикла и находятся все кратчайшие пути (если их несколько) между двумя заданными точками на графе. Подобный алгоритм можно также использовать, например, для поиска всех нужных файлов на диске. Относительно закрашивания растровых фигур, то здесь вершинами графа являются пикселы, а отметка пройденных пикселов делается прямо в растре цветом закрашивания. Как видим, это почти полностью повторяет идею предыдущего простейшего алгоритма, однако здесь мы не будем использовать рекурсию. Это обуславливает совсем другую последовательность обработки пикселов при закрашивании.

Запишем волновой алгоритм закрашивания на языке C++ с использованием графических функций API Windows.

```
void WaveFill(HDC hdc,int xst,int yst)
{
int num,num;
POINT *stackA, *stackB;

stackA = new POINT [10000];    //открываем массивы
stackB = new POINT [10000];    //для текущих фронтов волн
numA=1;
stackA[0]. x = xst;    //в массив stackA  записываем
stackA[0]. y = yst;    //координаты стартовой точки
numB=0;                //массив stackB пока что пуст
while ( 1 )            //основной цикл
  {
  if (numA > 0) OneStep(hdc,&numA,&numB,stackA,stackB);
  else break;
  if (numB > 0) OneStep(hdc,&numB,&numA,stackB,stackA);
  else break;
  }
delete [] stackB;
delete [] stackA;
}
//----одна итерация (фронт) распространения волны-------
//из массива Src[] читаются координаты пройденных точек-
//--для каждой точки находится соседняя и записывается--
//--в массив Dest[] - в этом массиве будет новый фронт--
```

```
void OneStep(HDC hdc,int *numSrc,int *numDest,
                    POINT *Src,POINT *Dest)
{
int x,y,i;

*numDest = 0;
for (i=0; i<*numSrc; i++)
    {
    x = Src[i].x;
    y = Src[i].y;
    NearPix(hdc,x+1, y, numDest, Dest);
    NearPix(hdc,x-1, y, numDest, Dest);
    NearPix(hdc, x, y+1, numDest, Dest);
    NearPix(hdc, x, y-1, numDest, Dest);
    }
}
void NearPix(HDC hdc, int x, int y,
             int *numStack, POINT *Stack)
{
if (GetPixel(hdc, x, y) ! = 0)
   {
   SetPixel(hdc,x,y,0);          //пиксел закрашивания
   Stack[*numStack]. x = x;      //координаты в массив
   Stack[*numStack]. y = y;
   (*numStack)++;     //количество элементов в массиве
   }
}
```

Здесь цвет закрашивания и цвет контура — черный цвет (код 0). Пример работы алгоритма приведен на рис. 3.12.

От начальной точки распространяется волна пикселов закрашивания в виде ромба. В одном цикле `OneStep` закрашиваются пикселы вдоль линии периметра ромба (или нескольких ромбов в зависимости от сложности фигуры). В качестве рабочих массивов для текущего сохранения координат пикселов фронтов волн использованы динамические массивы емкостью по 10 000 элементов. Максимальная емкость массивов обуславливается размерами контура и рассчитывается эмпирически.

Достаточно просто модифицировать приведенный алгоритм для случая отличающихся цветов контура и заполнения.

Необходимо заметить, что этот алгоритм не является самым быстрым из известных алгоритмов закрашивания, особенно если для его реализации ис-

пользовать медленную функцию `SetPixel` для рисования отдельных пикселов в программах для Windows. Большую скорость закрашивания обеспечивают алгоритмы, которые обрабатывают не отдельные пикселы, а сразу большие блоки из многих пикселов, которые образовывают прямоугольники или линии.

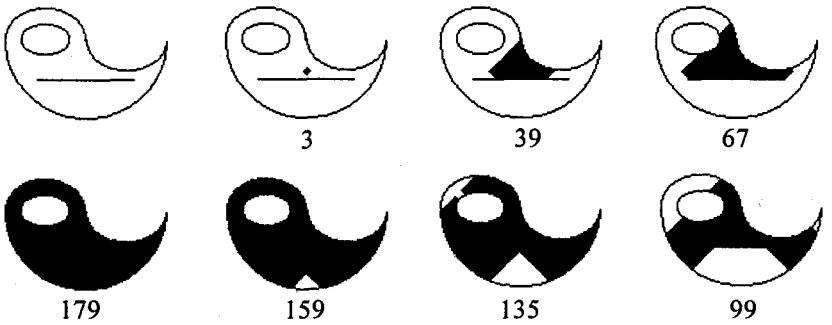


Рис. 3.12. Количество циклов `OneStep`

Алгоритм закрашивания линиями. Данный алгоритм получил широкое распространение в компьютерной графике. От приведенного ранее простейшего рекурсивного алгоритма он отличается тем, что на каждом шаге закрашивания рисуется горизонтальная линия, которая размещается между пикселами контура. Алгоритм рекурсивный, но поскольку вызов функции осуществляется для линии, а не для каждого отдельного пиксела, то количество вложенных вызовов уменьшается пропорционально длине линии. Это уменьшает нагрузку на стековую память компьютера.

Приведем запись алгоритма на языке C++. Этот пример взят из [32], незначительно усовершенствован и переработан для использования в среде Windows.

```
int LineFill(int x,int y,int dir,int preXL,int preXR)
{
int xl=x,xr=x;
COLORREF clr;

do
  clr = GetPixel(hdc,--xl,y);
while (clr ! = BORDER);          //BORDER - цвет контура
do
  clr = GetPixel(hdc,++xr,y);
```

```
while (clr ! = BORDER);
xl++;xr--; //левая и правая границы текущей горизонтали
MoveToEx(hdc,xl,y,NULL);    //рисуем горизонтальную
LineTo(hdc,xr+1,y);         //линию закрашивания
for (x=xl;x<=xr;x++)
    {
    clr = GetPixel(hdc,x,y+dir);
    if (clr != BORDER)
       x = LineFill(x,y+dir,dir,xl,xr);
    }
for (x=xl;x<preXL;x++)
    {
    clr = GetPixel(hdc,x,y-dir);
    if (clr != BORDER)
       x = LineFill(x,y-dir,-dir,xl,xr);
    }
for (x=preXR;x<xr;x++)
    {
    clr = GetPixel(hdc,x,y-dir);
    if (clr != BORDER)
       x = LineFill(x,y-dir,-dir,xl,xr);
    }
return xr;
}
```

В программах функция `LineFill` используется таким образом:

```
LineFill(xst, yst, 1, xst, xst);
```

Пример работы алгоритма закрашивания линиями приведен на рис. 3.13.

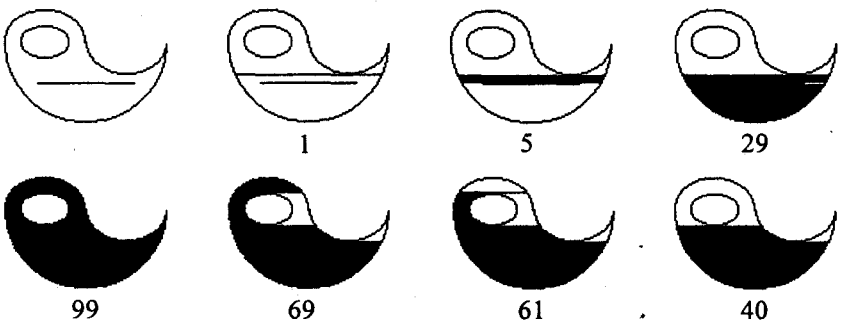


Рис. 3.13. Количество циклов `LineFill`

Так же, но немного быстрее, работает функция `FloodFill` API Windows. Разница в быстродействии обусловлена тем, что для `LineFill` мы использовали стандартные функции для интерфейса прикладных программ, а `FloodFill` оптимизирована на системном уровне.

Алгоритмы заполнения, которые используют математическое описание контура

Математическим описанием контура фигуры может служить уравнение $y = f(x)$ для окружности, эллипса или другой кривой. Для многоугольника (полигона) контур задается множеством координат вершин (x_i, y_i). Возможны и другие формы описания контура. В любом случае алгоритмы данного класса не предусматривают обязательное предварительное создание пикселов контура растра — контур может совсем не выводиться в растр. Рассмотрим некоторые из подобных алгоритмов заполнения.

Заполнение прямоугольников. Среди всех фигур прямоугольник заполнять наиболее просто. Если прямоугольник задан координатами противоположных углов, например, левого верхнего (x_1, y_1) и правого нижнего (x_2, y_2), тогда алгоритм может заключаться в последовательном рисовании горизонтальных линий заданного цвета (рис. 3.14).

```
for (y=y1; y<=y2; y++);
   //Рисуем горизонтальную линию
   //с координатами (x1, y)  - (x2, y)
```

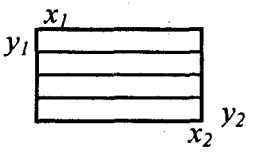


Рис. 3.14. Заполнение прямоугольника

Заполнение круга. Для заполнения круга можно использовать алгоритм вывода контура, который мы уже рассмотрели в разд. 3.2. В процессе выполнения этого алгоритма последовательно вычисляются координаты пикселов контура в границах одного октанта. Для заполнения надлежит выводить горизонтали, которые соединяют пары точек на контуре, расположенные симметрично относительно оси y (рис. 3.15).

Так же может быть построен и алгоритм заполнения эллипса.

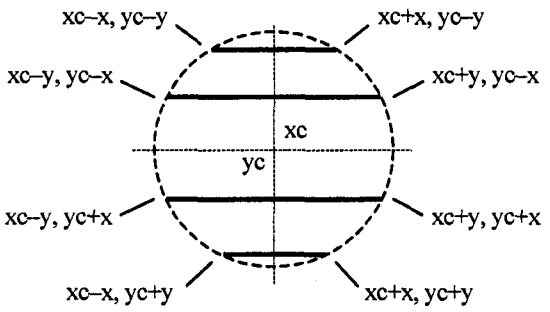


Рис. 3.15. Заполнение круга

Заполнение полигонов. Контур полигона определяется вершинами, которые соединены отрезками прямых (рис. 3.16). Это — векторная форма задания фигуры.

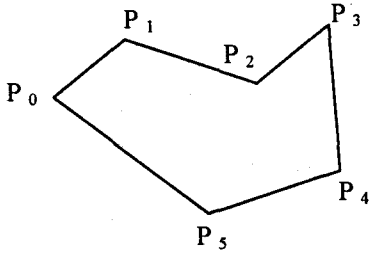


Рис. 3.16. Пример полигона

Рассмотрим один из наиболее популярных алгоритмов заполнения полигона. Его основная идея — закрашивание фигуры отрезками прямых линий. Наиболее удобно использовать горизонтали. Алгоритм представляет собою цикл вдоль оси *y*, в ходе этого цикла выполняется поиск точек сечения линии контура с соответствующими горизонталями. Этот алгоритм получил название XY [33]:

1. Найти **min**$\{y_i\}$ и **max**$\{y_i\}$ среди всех вершин P_i.
2. Выполнить цикл по y от y = **min** до y = **max** .
   {
   3. Нахождение точек пересечения всех отрезков контура с горизонталью y. Координаты x_j точек сечения записать в массив.
   4. Сортировка массива $\{ x_j \}$ по возрастанию x.

```
5. Вывод горизонтальных отрезков с координатами
                    (x0, y) - (x1, y)
                    (x2, y) - (x3, y)
                . . . . . . . . . . . . . . . .
                    (x2k, y) - (x2k+1, y)
   Каждый отрезок выводится цветом заполнения
}
```

В этом алгоритме использовано свойство топологии контура фигуры. Оно состоит в том, что любая прямая линия пересекает любой замкнутый контур четное количество раз (рис. 3.17). Для выпуклых фигур точек пересечения с любой прямой всегда две. Таким образом, на шаге 3 этого алгоритма в массив $\{ x_j \}$ всегда должно записываться парное число точек сечения.

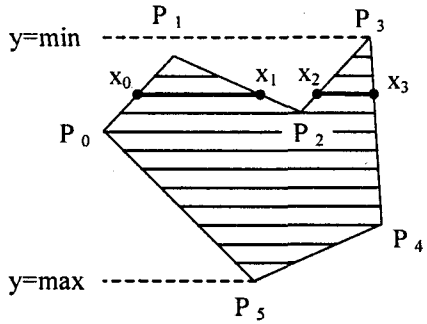


Рис. 3.17. Заполнение полигона

При нахождении точек пересечения горизонтали с контуром необходимо принимать во внимание особые точки. Если горизонталь имеет координату (y), совпадающую с y_i вершины P_i, тогда надлежит анализировать то, как горизонталь проходит через вершину. Если горизонталь при этом *пересекает* контур, как, например, в вершинах P_0 или P_4, то в массив записывается одна точка сечения. Если горизонталь *касается* вершины контура (в этом случае вершина соответствует локальному минимуму или максимуму как, например, в вершинах P_1, P_2, P_3 или P_5), тогда координата точки касания или не записывается, или записывается в массив два раза. Это условие четности количества точек пересечения, хранящихся в массиве $\{x_j\}$.

Процедура определения точек пересечения контура с горизонтальной разверткой, учитывая анализ на локальный максимум, может быть достаточно сложной. Это замедляет работу. Радикальным решением для упрощения поиска точек сечения может быть смещение координат вершин контура или горизонталей заполнения таким образом, чтобы ни одна горизонталь не по-

пала в вершину. Смещение можно выполнять различными способами, например, ввести в растр дробные координаты для горизонталей: $y_{min}+ 0.5$, $y_{min}+ 1.5$, ..., $y_{max} - 0,5$.

Но такое упрощение процедуры нахождения точек пересечения приводит к искажению формы полигона.

Для определения координат (x) точек пересечения для каждой горизонтали необходимо перебирать все n отрезков контура. Координата пересечения отрезка $p_i — p_k$ с горизонталью y равна

$$x = x_i + (y_k - y)(x_k - x_i) / (y_k - y_i).$$

Количество тактов работы этого алгоритма:

$$N_{тактов} \approx (y_{max} - y_{min}) N_{гор},$$

где y_{max}, y_{min} — диапазон координат y, $N_{гор}$ — число тактов, необходимых для одной горизонтали.

Оценим величину $N_{гор}$ как пропорциональную числу вершин

$$N_{гор} \approx k \cdot n,$$

где k — коэффициент пропорциональности, n — число вершин полигона.

Возможны модификации приведенного алгоритма для ускорения его работы. Например, можно принять во внимание то, что каждая горизонталь в большинстве случаев пересекает небольшое количество ребер контура. Поэтому если при поиске точек сечения делать предварительный отбор ребер, которые находятся вокруг каждой горизонтали, то можно добиться уменьшения количества тактов работы с $N_{гор} = k \cdot n$ до $k \cdot n_p$, где n_p — число отобранных ребер. Например, разделим диапазон $y_{min} - y_{max}$ пополам. Если для диапазона от y_{min} до $y_{ср}$ составить список отрезков (или вершин), которые попадают в этот диапазон, то в этот список будет включено приблизительно вдвое меньшее количество, чем для всего диапазона от y_{min} до y_{max}. Почему приблизительно — ибо это зависит от формы полигона. Таким образом, при работе алгоритма для каждой горизонтали в диапазоне y_{min} до $y_{ср}$ уже нужно не $k \cdot n$ тактов, а $\sim k \cdot n/2$.

Аналогично для диапазона $y_{ср} - y_{max}$ также можно составить список ребер, который также будет почти вдвое меньшим. Если принять, что подсчеты для каждой горизонтали теперь выполняются за вдвое меньшее количество тактов, а именно за ($N_{гор}/2$), то общее число тактов:

$$N_{тактов} \approx (y_{max} - y_{min}) N_{гор}/2 + N_{доп},$$

где $N_{доп}$ — количество тактов для создания списка ребер.

Такой способ повышения быстродействия эффективен для большого количества вершин. Контур можно делить не пополам, а на более мелкие части — соответственно повышается скорость.

Приведенные выше алгоритмы заполнения могут быть использованы не только для рисования фигур. На основе алгоритмов заполнения могут быть разработаны алгоритмы для других целей. Например, для определения площади фигуры, если считать площадь пропорциональной количеству пикселов заполнения. Или, например, алгоритм для поиска объектов по внутренней точке — эта операция часто используется в векторных графических редакторах.

3.6. Стиль линии. Перо

Что общее и что разное у объектов, изображенных на рис. 3.18?

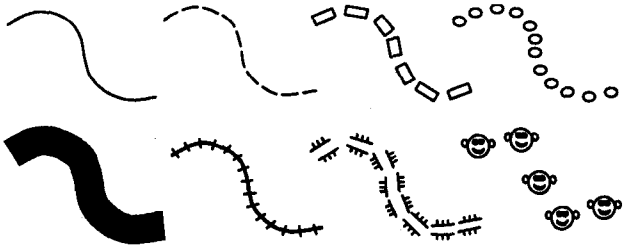


Рис. 3.18. Примеры стилей линий

Общее — линия оси, описывающая каждый из восьми объектов. Разное — элементы вдоль линии оси. Для описания различных по виду изображений на основе линий используют термин ***стиль линий*** или ***перо***. Термин ***перо*** иногда делает более понятной суть алгоритма вывода линий для некоторых стилей — в особенности для толстых линий. Например, если для тонкой непрерывной линии перо соответствует одному пикселу, то для толстых линий перо можно представить себе как фигуру или отрезок линии, который скользит вдоль оси линии, оставляя за собой след (табл. 3.1).

Таблица 3.1

Форма пера	—	|	/	■	●
Результат					

Алгоритмы вывода толстой линии

Взяв за основу любой алгоритм вывода обычных тонких линий (например, алгоритм Брезенхэма), запишем его в следующем обобщенном виде:

```
- - -
Вывод пиксела (x, y)
- - -
```

Можно представить себе такой алгоритм, как цикл, в котором определяются координаты (x,y) каждого пиксела. Этот алгоритм можно модифицировать для вывода толстой линии следующим образом:

```
- - - -
Вывод фигуры (или линии) пера с центром (x, y)
- - - -
```

Вместо вывода отдельного пиксела стоит вывод фигуры или линии, соответствующей перу — прямоугольник, круг, отрезок прямой.

Такой подход к разработке алгоритмов толстых линий имеет преимущества и недостатки. Преимущество — можно прямо использовать эффективные алгоритмы для вычисления координат точек линии оси, например, алгоритмы Брезенхэма. Недостаток — неэффективность для некоторых форм пера. Для перьев, которые соответствуют фигурам с заполнением, количество тактов работы алгоритма пропорционально квадрату толщины линии. При этом большинство пикселов многократно закрашивается в одних и тех же точках (рис. 3.19).

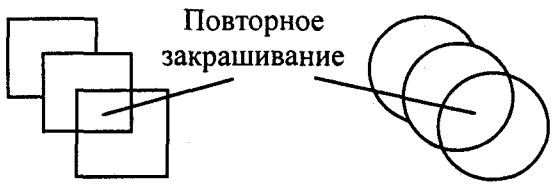


Рис. 3.19. Прямоугольное и круглое перья работают избыточно

Такие алгоритмы более эффективны для перьев в виде отрезков линий. В этом случае каждый пиксел рисуется только один раз. Но здесь важным является наклон изображаемой линии. Ширина пера зависит от наклона (рис. 3.20).

Очевидно, горизонтальное перо не может рисовать толстую горизонтальную линию.

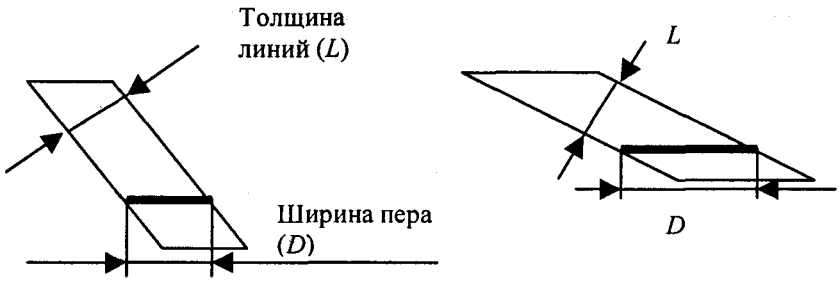


Рис. 3.20. Перья в виде отрезков линий

Для вывода толстых линий с помощью пера в качестве отрезка линии чаще всего используются отрезки горизонтальной или вертикальной линий, реже — диагональные отрезки под углом 45 градусов. Целесообразность использования такого способа определяется большой скоростью вывода горизонтальных и вертикальных отрезков прямой. Для того чтобы достигнуть минимального количества тактов вывода, толстые линии, которые по наклону ближе к вертикальным, рисуют горизонтальным пером, а пологие линии — вертикальным пером.

При выводе толстых линий с использованием пера-отрезка линии заметны разрывы в углах полилиний и плохие концы (рис. 3.21).

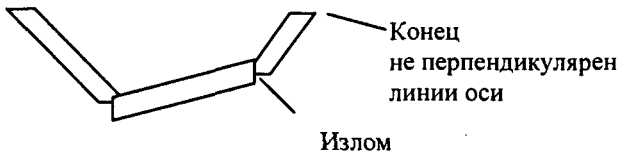


Рис. 3.21. Вывод толстых линий с использованием пера-отрезка

Для решения таких проблем иногда используют другие алгоритмы вывода толстых линий. Например, вывод толстой полилинии можно выполнить как рисование полигона с заполнением (рис. 3.22).

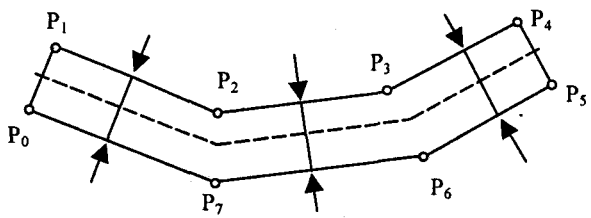


Рис. 3.22. Пример толстой полилинии

Очевидный недостаток такого подхода — меньшая скорость вывода, поскольку заполнение полигона — это существенно более трудоемкая задача, чем вывод линий, а кроме того, нужно еще определять координаты его вершин.

Третья разновидность алгоритмов вывода толстых линий — рисование толстой линии последовательным наложением нескольких тонких линий, смещенных одна относительно второй.

Алгоритмы вывода пунктирной линии

Алгоритм для рисования тонкой пунктирной линии можно получить из алгоритма вывода тонкой непрерывной линии

```
- - - -
Вывод пиксела (x, y)
- - - -
```

заменой процедуры вывода пиксела более сложной конструкцией:

```
- - - - -
Проверка значения счетчика C :
      Если C удовлетворяет некоторым условиям, то
            вывод пиксела (x, y)
Значение C увеличивается на единицу
- - - - -
```

В таком алгоритме используется новая переменная (C) — счетчик пикселов линии. Если значение C удовлетворяет некоторому логическому условию, то рисуется пиксел заданного цвета с текущими координатами (x, y). Логическое условие будет определять стиль линии. Например, если условием будет четность значения C, то получим линию из одиночных точек. Для рисования пунктирной линии можно анализировать остаток от деления C на S. Например, если рисовать пикселы линии только тогда, когда $C \bmod S < S/2$, то получим пунктирную линию с длиной штрихов $S/2$ и с шагом S.

При выводе полилиний, которые состоят из отрезков прямых, или сплайновых кривых, необходимо предотвратить обнуление значения счетчика в начале каждого отрезка и обеспечить продолжение непрерывного приращения вдоль всей сложной линии. Иначе будут нестыковки пунктира. Использование переменной-счетчика затруднено при генерации пунктирных линий в алгоритмах, которые используют симметрию, например, при выводе круга или эллипса. В этом случае будут нестыковки пунктира на границах октантов или квадрантов.

Алгоритм вывода толстой пунктирной линии

Объединив алгоритм для вывода толстой непрерывной линии и алгоритм для тонкой пунктирной линии, можно получить следующий алгоритм:

```
- - - -
Проверка значения счетчика (C) :
    Если (C) удовлетворяет условию , то
            вывод фигуры (линии) пера с центром в (x, y)
C = C + 1;
- - - -
```

Такой алгоритм достаточно прост. На практике используются и более изощренные алгоритмы.

3.7. Стиль заполнения. Кисть. Текстура

При выводе фигур могут использоваться различные стили заполнения. Для обозначения стилей заполнения, отличных от сплошного стиля, используют такие понятия, как ***кисть*** и ***текстура***. Их можно считать синонимами, однако понятие текстуры обычно используется применительно к трехмерным объектам, а кисть — для изображения двумерных объектов. Текстура — это стиль заполнения, закрашивание, которое имитирует сложную рельефную объемную поверхность, выполненную из какого-то материала.

Для описания алгоритмов заполнения фигур с определенным стилем используем тот же способ, что и для описания алгоритмов рисования линий. Мы уже ранее рассмотрели некоторые алгоритмы заполнения, и вы, наверное, согласитесь, что описание всех разновидностей подобных алгоритмов можно дать с помощью такой обобщенной схемы:

```
- - - -
Вывод пиксела заполнения цвета C с координатами (x, y)
- - - -
```

Например, в алгоритме вывода полигонов пикселы заполнения рисуются в теле цикла горизонталей, а все другие операции предназначены для подсчета координат (x, y) этих пикселов. Сплошное заполнение означает, что цвет (C) всех пикселов одинаков, то есть C=const. Нам нужно как-то изменять цвет пикселов заполнения, чтобы получить определенный узор. Преобразуем алгоритм заполнения следующим образом:

```
- - - -
C = f (x, y)
Вывод пиксела заполнения (x, y) цветом C
- - - -
```

Функция $f(x,y)$ будет определять стиль заполнения. Аргументами функции цвета являются координаты текущего пиксела заполнения. Однако в отдельных случаях эти аргументы не нужны. Например, если цвет C вычислять как случайное значение в определенных границах: C = random (), то можно создать иллюзию шершавой матовой поверхности (рис. 3.23).

Рис. 3.23. Матовая поверхность

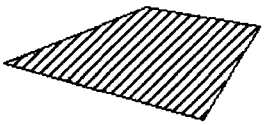


Рис. 3.24. Штриховка

Другой стиль заполнения — штриховой (рис. 3.24). Для него функцию цвета также можно записать в аналитической форме:

$$f(x, y) = \begin{cases} C_ш, & \text{если } (x + y) \bmod S < T, \\ C_ф & \text{— в других случаях,} \end{cases}$$

где S — период, а T — толщина штрихов, $C_ш$ — цвет штрихов, $C_ф$ — цвет фона.

Если не рисовать пикселы фона, то можно создать иллюзию полупрозрачной фигуры. Подобную функцию можно записать и для других типов штриховки. Аналитическая форма описания стиля заполнение позволяет достаточно просто изменять размеры штрихов при изменениях масштаба показа, например, для обеспечения режима WYSIWYG.

Зачастую при использовании кистей и текстур используется копирование небольших растровых изображений. Такой алгоритм заполнения можно описать вышеупомянутой общей схемой, если строку $C = f(x, y)$ заменить двумя другими строками:

```
- - -
Координаты пиксела заполнения (x, y) преобразуем в растровые
координаты образца кисти (xт, yт)
По координатам (xт, yт) определяем цвет (C) пиксела в образце кисти
Вывод пиксела заполнения цвета C с координатами (x, y)
- - -
```

Преобразование координат пиксела заполнения (x, y) в координаты внутр образца кисти можно выполнить таким образом:

$$x_T = x \bmod m,$$
$$y_T = y \bmod n,$$

где m, n — размеры растра кисти соответственно по горизонтали и вертикал При этом координаты (x_T, y_T) будут в диапазоне $x_T = 0...m - 1$, $y_T = 0...n -$ при любых значениях x и y. Таким образом, обеспечивается циклическое ко пирование фрагментов кисти внутри области заполнения фигуры (рис. 3.25).

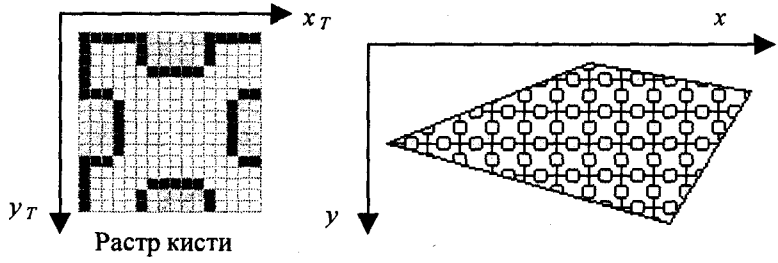


Рис. 3.25. Копирование растра кисти

Удобно, когда размеры кисти равны степени двойки. В этом случае вмест операций взятия остатка (mod) можно использовать более быстродействую щие для цифровых компьютеров поразрядные двоичные операции. Приведем пример вычисления остатка от деления на 16.

Биты двоичного кода числа X:

```
x x . . . x x x x x
```

```
X mod 16  =  0 0 . . . 0 x x x x
```

Здесь можно использовать поразрядную операцию & (И).

Еще один пример. Если необходимо вычислить **X mod 256**, а значение **X** за писано в регистре **AX** микропроцессора (совместимого с 80×86), то в качест ве результата достаточно взять содержимое младшей байтовой части этого регистра — **AL**.

Для пикселов текстур часто употребляется название ***тексел***.

Растровые текстуры и кисти широко используются в современной компью терной графике, в том числе и в 3D-графике. Для отображения трехмерных

объектов часто используются полигональные поверхности, каждая грань отображается с наложенной текстурой. Поскольку объекты обычно показываются с разных ракурсов — повороты, изменения размеров и тому подобное, то необходимо соответственно трансформировать и каждую грань с текстурой. Для этого используются ***проективные текстуры***.

Общая схема алгоритма заполнения контуров полигонов для проективных текстур такая же, как и приведенная выше. Однако растровый образец здесь представляет всю грань, а преобразование координат из (x, y) в (x_T, y_T) более сложное, например, аффинное:

$$x_T = A\,x + B\,y + C,$$
$$y_T = D\,x + E\,y + F,$$

где коэффициенты $A, B, \ldots, F$ — константы при пересчете координат всех пикселов для отдельной текстурированной грани. Такое преобразование координат можно использовать, если привязать текстуру к грани по трем опорным точкам. Пример наложения проективной текстуры приведен на рис. 3.26.

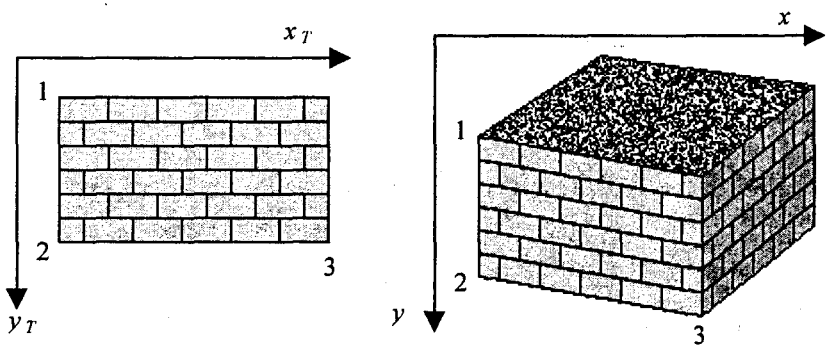


Рис. 3.26. Наложение проективной текстуры на две грани объекта

Привязывание по трем точкам соответствует уравнениям:

$$x_{Ti} = A\,x_i + B\,y_i + C,$$
$$y_{Ti} = D\,x_i + E\,y_i + F,$$

где $i = 1, 2, 3$. По известным координатам (x_{Ti}, y_{Ti}) и (x_i, y_i) можно найти коэффициенты $A, B, \ldots, F$, если решить систему линейных уравнений. Эта система распадается на две независимые системы третьего порядка. Для упрощения записи заменим x_{Ti} на X_i, а y_{Ti} на Y_i:

$$X_1 = A x_1 + B y_1 + C,$$
$$X_2 = A x_2 + B y_2 + C,$$
$$X_3 = A x_3 + B y_3 + C,$$

и

$$Y_1 = D x_1 + E y_1 + F,$$
$$Y_2 = D x_2 + E y_2 + F,$$
$$Y_3 = D x_3 + E y_3 + F.$$

Для решения систем линейных уравнений известно множество способов. пользуем способ, основанный на вычислении определителей. Решение вой системы для коэффициентов A, B и C можно записать в виде

$$A = detA / det,$$
$$B = detB / det,$$
$$C = detC / det,$$

где $det = \begin{Vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{Vmatrix}$ — это главный определитель системы, а опре

лители $detA$, $detB$ и $detC$ получаются заменой соответствующих столб в det столбцом свободных членов

$$detA = \begin{Vmatrix} X_1 & y_1 & 1 \\ X_2 & y_2 & 1 \\ X_3 & y_3 & 1 \end{Vmatrix}, \quad detB = \begin{Vmatrix} x_1 & X_1 & 1 \\ x_2 & X_2 & 1 \\ x_3 & X_3 & 1 \end{Vmatrix}, \quad detC = \begin{Vmatrix} x_1 & y_1 & X_1 \\ x_2 & y_2 & X_2 \\ x_3 & y_3 & X_3 \end{Vmatrix}$$

Если главный определитель равен нулю, то это означает, что решение сис мы невозможно. Это может быть, например, тогда, когда все три точ (x_1, y_1), (x_2, y_2) и (x_3, y_3) располагаются вдоль прямой линии (грань вид с торца). Однако в этом случае рисовать текстуру и не нужно. Вычислить определитель третьей степени можно, например, по "правилу Саррюса" [3]. Д этого справа нужно дописать первые два столбца, а затем сложить (вычес произведения по диагоналям:

$$\begin{Vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{Vmatrix} = \begin{Vmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{Vmatrix} =$$

$$- \quad - \quad - \quad + \quad + \quad +$$

$$= a_{11} a_{22} a_{33} + a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32} - a_{13} a_{22} a_{31} - a_{11} a_{23} a_{32} - a_{12} a_{21} a_{33}.$$

Вычислим главный определитель

$$det = \begin{Vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{Vmatrix} = x_1 y_2 + y_1 x_3 + x_2 y_3 - y_2 x_3 - x_1 y_3 - y_1 x_2.$$

Преобразуем выражение так, чтобы уменьшить число умножений:

$$det = x_1 (y_2 - y_3) + x_2 (y_3 - y_1) + x_3 (y_1 - y_2).$$

Аналогично вычисляются определители *detA* и *detB*. Определитель *detC* является самым сложным из всех. Но его вычислять не обязательно. Запишем решение системы в следующем виде:

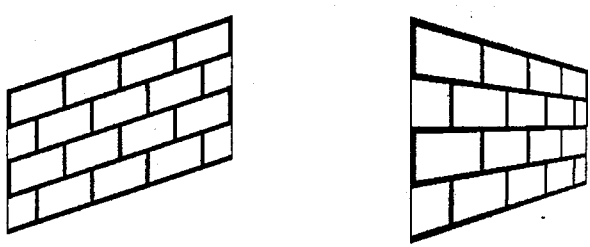
$$A = detA / det = (X_1 (y_2 - y_3) + X_2 (y_3 - y_1) + X_3 (y_1 - y_2)) / det,$$
$$B = detB / det = (x_1 (X_2 - X_3) + x_2 (X_3 - X_1) + x_3 (X_1 - X_2)) / det,$$
$$C = X_1 - A x_1 - B y_1 .$$

Таким же способом решаем систему уравнений для D, E и F.

$$D = (Y_1 (y_2 - y_3) + Y_2 (y_3 - y_1) + Y_3 (y_1 - y_2)) / det;$$
$$E = (x_1 (Y_2 - Y_3) + x_2 (Y_3 - Y_1) + x_3 (Y_1 - Y_2)) / det;$$
$$F = Y_1 - D x_1 - E y_1 .$$

Заметьте, что здесь главный определитель *det* совпадает с определителем первой системы уравнений — для A, B и C.

Наложение текстур в перспективной проекции сложнее, чем для аксонометрической проекции. Рассмотрим рис. 3.27, на котором изображен текстурированный прямоугольник.



Аксонометрическая проекция

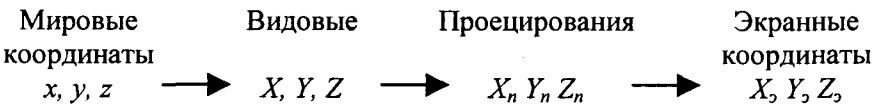
Перспективная

Рис. 3.27. Прямоугольник в различных проекциях

Прямоугольник в аксонометрической (параллельной) проекции всегда выглядит как параллелограмм, поскольку для этой проекции сохраняется параллельность прямых и отношение длин. В перспективной (центральной) проекции это уже не параллелограмм и не трапеция (в косоугольной — трапеция), поскольку параллельность и отношение длин здесь не сохраняются. А что сохраняется? Как изображать плоские грани?

В этой книге мы рассматриваем проекции на плоскость. Для таких проекций прямые линии остаются прямыми линиями, поэтому грани можно выводить как полигоны.

Здесь уместно вспомнить, как формируется изображение в некоторой проекции средствами компьютерной графики. Как уже нами было рассмотрено в главе 2, последовательность преобразований координат выглядит так:



Если считать, что точки текстуры должны соответствовать точкам на объекте, то координаты текстуры должны связываться с мировыми координатами. Однако поскольку для аксонометрической проекции в цепочке от мировых координат до экранных все преобразования ***линейны***, то вполне допустимо связать координаты текстуры с экранными координатами одним аффинным преобразованием.

Для перспективной проекции так делать нельзя. Преобразование координат из видовых координат в координаты плоскости проецирования ***не линейно***. Поэтому экранные координаты вначале следует преобразовать в такие, которые линейно связаны с мировыми — это могут быть, скажем, видовые. А затем видовые координаты (или непосредственно мировые) связать с координатами текстуры аффинным преобразованием, используя, например, метод трех точек.

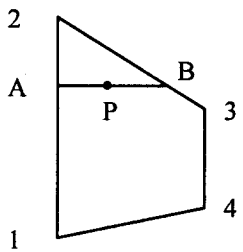


Рис. 3.28. Полигон

Рассмотрим, как можно выводить в перспективной проекции полигон с текстурой. Будем использовать алгоритм заполнения полигона горизонтальными линиями, уже рассмотренный нами. На рис. 3.28 изображена одна из горизонталей (AB). Вершины полигона (1-2-3-4) здесь заданы экранными двумерными координатами. Для краткости изложения положим, что экранные координаты совпадают с координатами в плоскости проецирования. В ходе вы-

вода полигона для связи с текстурой будем вычислять видовые координаты произвольной точки (P) этого полигона. Для этого будем использовать в качестве базовой такую операцию: по известным видовым координатам концов отрезка находим видовые координаты точки отрезка, заданной координатами в плоскости проецирования.

Для определения видовых координат X, Y, Z точки A должны быть известны видовые координаты концов отрезка (1–2). Тогда справедливы соотношения:

$$\frac{X-X_1}{X_2-X_1}=\frac{Y-Y_1}{Y_2-Y_1}=\frac{Z-Z_1}{Z_2-Z_1}.$$

Выберем пропорцию, связывающую координаты X и Z. Тогда

$$X = a + Zb,$$

где $a = X_1 - Z_1 (X_2 - X_1)/(Z_2 - Z_1)$, $b = (X_2 - X_1)/(Z_2 - Z_1)$.

Теперь запишем для перспективной проекции соотношение между видовыми координатами произвольной точки и координатой X_n в плоскости проецирования:

$$X_n = X\frac{Z_к - Z_{пл}}{Z_к - Z},$$

где $Z_к$ — это координата камеры (точки схода лучей проектирования), $Z_{пл}$ – координата плоскости проецирования. Перепишем это равенство так:

$$X = c + Zd,$$

где $c = X_n Z_к / (Z_к - Z_{пл})$, $d = - X_n / (Z_к - Z_{пл})$. Теперь решим систему уравнений:

$$X = a + Zb,$$
$$X = c + Zd.$$

Решением системы будет:

$$Z = \frac{c-a}{b-d},$$

после чего вычисляется X, например, по формуле $X = a + Zb$. Для определения координаты Y достаточно заменить везде X на Y. Здесь можно отметить, что вычисление видовых координат (X, Y, Z) соответствует *обратному проективному преобразованию*.

Найдя видовые координаты точки A, мы можем точно так же вычислить видовые координаты и для точки B, лежащей на отрезке (3-4). Аналогично вычисляются видовые координаты точки P.

Следует отметить, что, несмотря на то, что для преобразования координат необходимо вычислять дробно-линейные выражения, цикл вычислений можно сделать достаточно простым подобно инкрементным алгоритмам. Сделайте это самостоятельно в виде упражнения.

Для точного наложения текстур на поверхности используются и более сложные преобразования координат. Некоторые из них нами будут рассмотрены в главе 5.

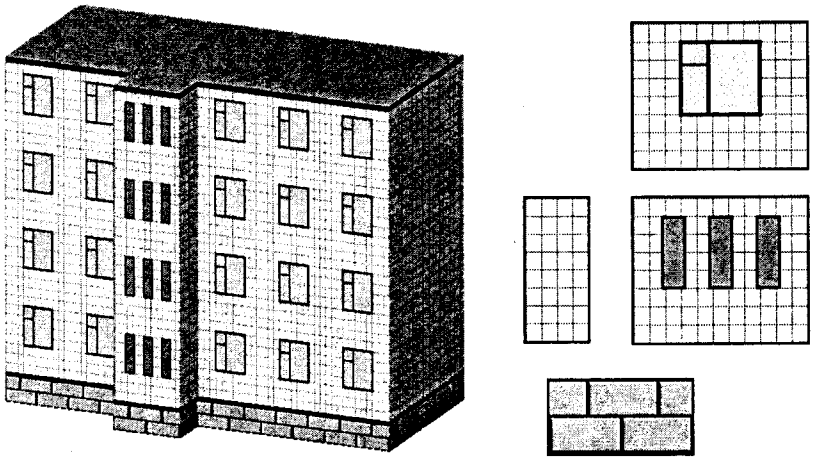
Одна из проблем наложения текстур заключается в том, что преобразование растровых образцов (повороты, изменение размеров и тому подобное) приводят к ухудшению качества растров. Повороты растра добавляют ступенчатость (*aliasing*); увеличение размеров укрупняет пикселы, а уменьшение размеров растра приводит к потере многих пикселов образца текстуры, появляется муар. Для улучшения текстурованных изображений используют методы фильтрации (интерполяции) растров текстур [41, 47]. Также используются несколько образцов текстур для различных ракурсов показа (*mipmaps*) — компьютерная система во время отображения находит в памяти наиболее пригодный растровый образец.

Для использования текстур необходим достаточный объем памяти компьютера — количество растровых образцов может достигать десятков, сотен и более в зависимости от количества типов объектов и многообразия пространственных сцен. Чтобы как можно быстрее создавать изображение, необходимо сохранять текстуры в оперативной памяти.

Для экономии памяти, выделяемой для текстур, можно использовать блочное текстурирование. Текстура здесь уже не представляет всю грань целиком, а лишь отдельный фрагмент, который циклически повторяется в грани. Это напоминает процесс размножения рисунка кисти при закрашивании полигонов, рассмотренный нами в этом разделе.

Разумеется, далеко не для всех объектов можно использовать такой способ отображения, однако, например, для образцов современной массовой "коробочной" архитектуры в этом плане имеются практически неограниченные возможности (рис. 3.29).

Современные видеоадаптеры оснащены графическими процессорами, которые аппаратно поддерживают операции с текстурами.



Изображение в проекции Использованные текстуры

Рис. 3.29. Блочное текстурирование

3.8. Фракталы

Фрактал можно определить как объект довольно сложной формы, получающийся в результате выполнения простого итерационного цикла. Итерационность, рекурсивность обуславливают такие свойства фракталов, как *самоподобие* — отдельные части похожи по форме на весь фрактал в целом. Латинское **fractus** означает "составленный из фрагментов". В 1975 году французский математик Бенуа Мандельброт издал книгу "The fractal Geometry of Nature". С того времени слово "фрактал" стало модным.

Фракталом Мандельброта названа фигура, которая порождается очень простым циклом. Для создания этого фрактала необходимо для каждой точки изображения выполнить цикл итераций согласно формуле:

$$z_{k+1} = z_k^2 + z_0,$$

где $k = 0, 1, \ldots, n$. Величины z_k — это комплексные числа, $z_k = x_k + \boldsymbol{i}\, y_k$, причем стартовые значения x_0 и y_0 — это координаты точки изображения. Для каждой точки изображения итерации выполняются ограниченное количество раз (n) или до тех пор, пока модуль числа z_k не превышает 2. Модуль комплексного числа равен корню квадратному из x^2+y^2. Для вычисления квадрата величины z_k можно воспользоваться формулой $z = (x + \boldsymbol{i}y)\,(x + \boldsymbol{i}y) = (x^2 - y^2) + \boldsymbol{i}\,(2y)$, поскольку $\boldsymbol{i}^2 = -1$. Цикл итераций для фрактала Мандель-

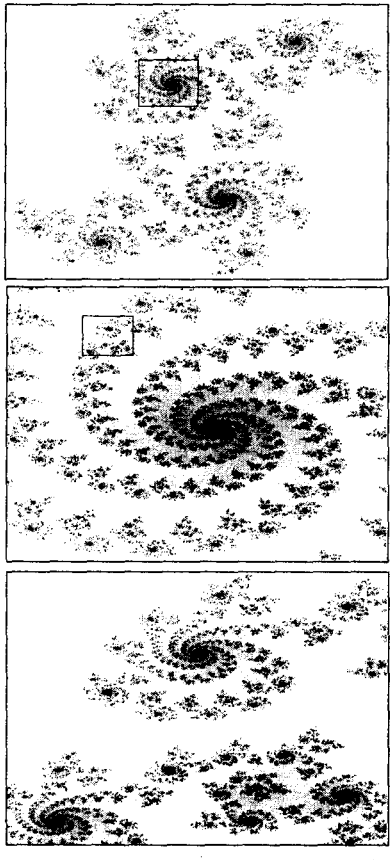


Рис. 3.30. Фрактал Джулия

брота можно выполнять в диапазон x = (от –2.2 до 1), y = (от –1.2 до 1.2). Для то го чтобы получить изображение в растре, не обходимо пересчитывать координаты этог диапазона в пикселные. В главе 6 мы рас смотрим соответствующий пример програм мы, генерирующей изображения этого фрак тала.

Фрактал Джулия внешне совсем не похож н фрактал Мандельброта, однако он определя ется итерационным циклом, почти полностьк тождественным с циклом генерации Ман дельброта. Формула итераций для фрактала Джулия такая:

$$z_{k+1} = z_k^2 + c,$$

где c — комплексная константа. Условием завершения итераций является $| z_k | > 2$ — так же, как для фрактала Мандельброта.

На рис. 3.30 приведено несколько изображений фрактала Джулия для $c = 0.36 + i\ 0.36$, n=256. На верхнем образце картинка построена в границах x = (от –1 до 1), y = (от –1.2 до 1.2). На втором и третьем изображениях рис. 3.30 показаны увеличенные фрагменты фрактала. Как видим, фрактал самоподобный — при любом увеличении отдельные части напоминают формы целого. Самоподобие считается важным свойством фракталов. Это отличает их от других типов объектов сложной формы.

Рассмотрим следующий пример фрактала — фрактал Ньютон. Для него итерационная формула имеет такой вид:

$$z_{k+1} = \frac{3z_k^4 + 1}{4z_k^3},$$

где z — также комплексные числа, причем $z_0 = x + iy$ соответствует координатам точки изображения.

Условием прекращения цикла итераций для фрактала Ньютон есть приближение значений $|z^4 - 1|$ к нулю. Например, изображение на рис. 3.31 было получено для $|z^4 - 1|^2 > 0.001$, границы расчета x = (от –1 до 1), y = (от –1 до 1).

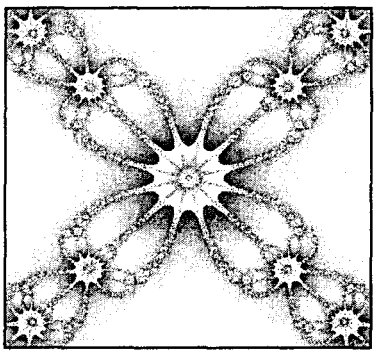


Рис. 3.31. Фрактал Ньютон

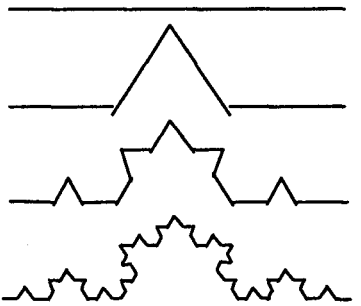


Рис. 3.32. Геометрические итерации для кривой Кох

Рассмотрим еще одну разновидность фракталов. Такие фракталы названы геометрическими, поскольку их форма может быть описана как последовательность простых геометрических операций. Например, кривая Кох становится фракталом в результате бесконечного количества итераций, в ходе которых выполняется деление каждого отрезка прямой на три части. На рис. 3.32 показаны три итерации — постепенно линия становится похожей на снежинку.

Следующую группу составляют фракталы, которые генерируются согласно методу "систем итеративных функций" — IFS (Iterated Functions Systems). Этот метод может быть описан, как последовательный итеративный расчет координат новых точек в пространстве:

$$x_{k+1} = F_x(x_k, y_k),$$
$$y_{k+1} = F_y(x_k, y_k),$$

где F_x и F_y — функции преобразования координат, например, аффинного преобразования. Эти функции и определяют форму фрактала. В случае аффинного преобразования необходимо найти соответствующие числовые значения коэффициентов.

Давайте попробуем создать фрактал, который бы выглядел как растение. Вообразим себе ствол, на котором много веточек. На каждой веточке много меньших веточек и так далее. Самые малые ветки можно считать листвой или колючками. Все элементы будем рисовать отрезками прямой. Каждый отрезок будет задаваться двумя конечными точками.

Для начала итераций необходимо задать стартовые координаты линии. Это будут точки 1, 2. На каждом шаге итераций будем рассчитывать координаты других точек.

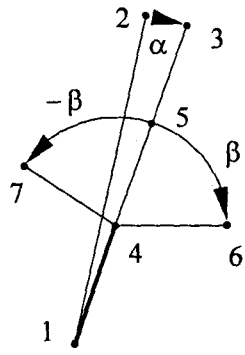


Рис. 3.33. Опорные точки элементов фрактала

Сначала находим точку 3. Это повернутая н угол α точка 2, центр поворота — в точке 1 (рис. 3.33)

$$x3 = (x2 - x1)\cos\alpha - (y2 - y1)\sin\alpha + x1,$$
$$y3 = (x2 - x1)\sin\alpha + (y2 - y1)\cos\alpha + y1.$$

Если $\alpha = 0$, то ствол и все ветви прямые. Потом на ходим точку 4. От нее будут распространяться ветви Пусть соотношение длин отрезков 1-4 и 1-3 равно k причем $0<k<1$. Тогда для вычисления координат точки 4 можно воспользоваться следующими формулами:

$$x4 = x1\,(1 - k) + x3\,k,$$
$$y4 = y1\,(1 - k) + y3\,k.$$

Теперь зададим длину и угол наклона ветвей, которые выходят из точки 4. Сначала найдем координаты точки 5. Введем еще один параметр — $k1$, который будет определять соотношение длин отрезков 4-5 и 4-3, причем так же $0<k1<1$. Координаты точки 5 равны:

$$x5 = x4\,(1 - k1) + x3\,k1,$$
$$y5 = y4\,(1 - k1) + y3\,k1.$$

Точки 6 и 7 — это точка 5, повернутая относительно точки 4 на углы β и $-\beta$ соответственно:

$$x6 = (x5 - x4)\cos\beta - (y5 - y4)\sin\beta + x4;$$
$$y6 = (x5 - x4)\sin\beta + (y5 - y4)\cos\beta + y4;$$
$$x7 = (x5 - x4)\cos\beta + (y5 - y4)\sin\beta + x4;$$
$$y7 = -(x5 - x4)\sin\beta + (y5 - y4)\cos\beta + y4.$$

Кроме расчета опорных точек на каждом шаге будем рисовать один отрезок 1-4. В зависимости от номера итераций можно изменять цвет отрезка. Также можно устанавливать его толщину, например, пропорционально длине.

Таким образом, фрактал мы задали как последовательность аффинных преобразований координат точек. Величины α, β, k, $k1$ — это параметры, которые описывают вид фрактала в целом. Они представляют собой константы на протяжении всего итеративного процесса. Это дает возможность в итерациях использовать только операции сложения, вычитания и умножения, если вы-

числить значения sin(), cos(), $(1 - k)$ и $(1 - k1)$ только один раз перед началом итераций как коэффициенты-константы.

Дадим запись алгоритма в виде рекурсивной процедуры ШАГ().

```
ШАГ(x1, y1, x2, y2, num)
{
Если (x1-x2)^2 + (y1-y2)^2 > lmin,
то
  {
  Вычисляем координаты точек 3-7:
     x3 = (x2 - x1) A -  (y2 - y1) B + x1
     y3 = (x2 - x1) B + (y2 - y1) A + y1
     x4 = x1 C + x3 D
     y4 = y1 C + y3 D
     x5 = x4 E + x3 F
     y5 = y4 E + y3 F
     x6 = (x5 -x4) G - (y5 - y4) H + x4
     y6 = (x5 -x4) H + (y5 - y4) G + y4
     x7 =   (x5 -x4) G + (y5 - y4) H + x4
     y7 = - (x5 -x4) H + (y5 - y4) G + y4
  Рисуем отрезок (x1,y1 - x4,y4)
  ШАГ(x4, y4, x3, y3, num)
  ШАГ(x4, y4, x6, y6, num+1)
  ШАГ(x4, y4, x7, y7, num+1)
  }
}
```

Для того чтобы нарисовать фрактал, необходимо вызывать процедуру ШАГ, установив соответствующие значения ее аргументов: ШАГ(x1, y1, x2, y2, 0). Обратите внимание на один из аргументов этой процедуры — num, который в начале работы равен 0. В теле процедуры есть три рекурсивных вызова с различными значениями этого аргумента:

ШАГ(x4, y4, x3, y3, num) — продолжаем ствол;
ШАГ(x4, y4, x6, y6, num+1) — правая ветвь;
ШАГ(x4, y4, x7, y7, num+1) — левая ветвь.

Значение num показывает степень детализации расчета дерева. Один цикл итераций содержит много шагов, соответствующих одному значению вели-

чины num. Числовое значение num можно использовать для прекращения итеративного процесса, а также для определения текущего цвета элементов "растения".

Завершение циклов итераций в нашем алгоритме происходит тогда, когда длина ветви становится меньше некоторой величины l_{min}, например, $l_{min}=1$. Этот фрактал при $\alpha = 2°$, $\beta = 86°$, $k = 0.14$, $k1 = 0.3$ похож на папоротник (рис. 3.34).

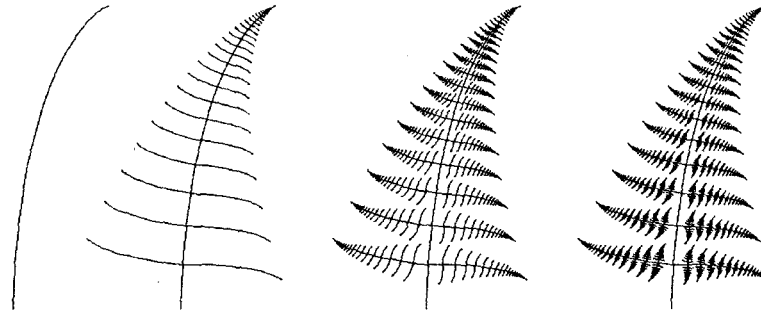
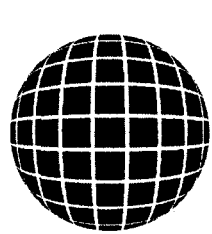


Рис. 3.34. Вид фрактала для разного количества циклов итераций

Метод IFS применяется не только для создания изображений. Его использовали *Барнсли* и *Слоан* для эффективного сжатия графических изображений при записи в файлы. Основная идея такая: поскольку фракталы могут представлять очень сложные изображения с помощью простых итераций, то описание этих итераций требует значительно меньшего объема информации, чем соответствующие растровые изображения. Для кодирования изображений необходимо решать обратную задачу — для изображения (или его фрагмента) подобрать соответствующие коэффициенты аффинного преобразования. Этот метод используется для записи цветных фотографий в файлы со сжатием в десятки и сотни раз без заметного ухудшения изображения. Формат таких графических файлов назван FIF (Fractal Image Format) и запатентован фирмой Iterated Systems [36].



ГЛАВА 4

Методы и алгоритмы трехмерной графики

Понятие "трехмерная графика" в настоящее время можно считать наиболее распространенным для обозначения темы, которую мы рассмотрим (в литературе название часто сокращается до "3D-графики"). Однако необходимо отметить, что такое название не совсем точно, ибо речь пойдет о создании изображения на плоскости, а не в объеме. Истинно трехмерные способы отображения пока что не достаточно широко распространены.

4.1. Модели описания поверхностей

Рассмотрим, как можно представлять форму трехмерных объектов в системах КГ. Для описания формы поверхностей могут использоваться разнообразные методы. Сделаем обзор некоторых из них.

Аналитическая модель

Аналитической моделью будем называть описание поверхности математическими формулами. В КГ можно использовать много разновидностей такого описания. Например, в виде функции двух аргументов $z = f(x, y)$. Можно использовать уравнение $F(x, y, z) = 0$.

Зачастую используется параметрическая форма описания поверхности. Запишем формулы для трехмерной декартовой системы координат (x, y, z):

$$x = F_x(s, t),$$
$$y = F_y(s, t),$$
$$z = F_z(s, t),$$

где s и t — параметры, которые изменяются в определенном диапазоне, функции F_x, F_y и F_z будут определять форму поверхности.

Преимущества параметрического описания — легко описывать поверхност которые отвечают неоднозначным функциям, замкнутые поверхности. Оп сание можно сделать таким образом, что формула не будет существенно и меняться при поворотах поверхности, масштабировании.

В качестве примера рассмотрим аналитическое описание поверхности шар Сначала как функцию двух аргументов:

$$z = \pm\sqrt{R^2 - x^2 - y^2}.$$

Затем в виде уравнения: $x^2 + y^2 + z^2 - R^2 = 0.$

А также в параметрической форме:

$$x = R \sin s \cos t,$$
$$y = R \sin s \sin t,$$
$$z = R \cos s.$$

Для описания сложных поверхностей часто используют *сплайны*. Сплайн – это специальная функция, более всего пригодная для аппроксимации отдель ных фрагментов поверхности. Несколько сплайнов образовывают модел сложной поверхности. Другими словами, сплайн — эта тоже поверхность, но такая, для которой можно достаточно просто вычислять координаты ее точек Обычно используют кубические сплайны. Почему именно кубические? По тому, что третья степень — наименьшая из степеней, позволяющих описы вать любую форму, и при стыковке сплайнов можно обеспечить непрерыв ную первую производную — такая поверхность будет без изломов в места стыка. Сплайны часто задают параметрически. Запишем формулу для компо ненты $x(s,t)$ кубического сплайна в виде многочлена третьей степени пара метров s и t:

$$\begin{aligned} x(s,t) = {} & a_{11} s^3 t^3 + a_{12} s^3 t^2 + a_{13} s^3 t + a_{14} s^3 + \\ & + a_{21} s^2 t^3 + a_{22} s^2 t^2 + a_{23} s^2 t + a_{24} s^2 + \\ & + a_{31} s t^3 + a_{32} s t^2 + a_{33} s t + a_{34} s + \\ & + a_{41} t^3 + a_{42} t^2 + a_{43} t + a_{44}. \end{aligned}$$

В математической литературе можно ознакомиться со способами определения коэффициентов a_{ij} для сплайнов, которые имеют заданные свойства. Приме ры анализа и синтеза сплайнов в матричной форме приведены в [19, 28].

Рассмотрим одну из разновидностей сплайнов — сплайн Безье. Приведем его сначала в обобщенной форме — степени $m \times n$ [19]:

$$P(s,t) = \sum_{i=0}^{m}\sum_{j=0}^{n} C_m^i s^i (1-s)^{m-i} C_n^j t^j (1-t)^{n-j} P_{ij},$$

где P_{ij} — опорные точки-ориентиры, $0 \le s \le 1$, $0 \le t \le 1$, C_m^i и C_n^j — коэффициенты бинома Ньютона, они рассчитываются по формуле:

$$C_a^b = \frac{a!}{b!(a-b)!}.$$

Кубический сплайн Безье соответствует значениям $m = 3$, $n = 3$. Для его определения необходимо 16 точек-ориентиров P_{ij} (рис. 4.1); коэффициенты C_m^i, C_n^j равны 1,3,3,1 при $i, j = 0,1,2,3$.

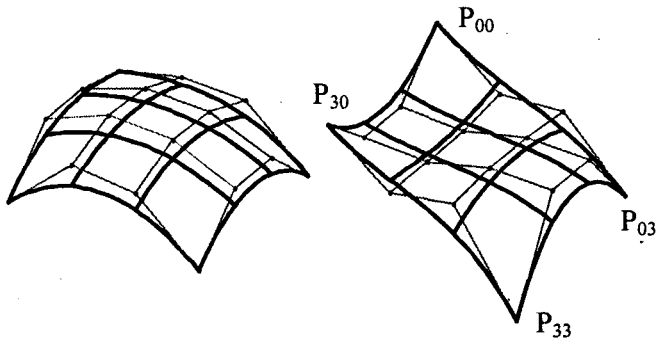


Рис. 4.1. Кубические сплайны Безье

Характеризуя аналитическую модель в целом, можно сказать, что эта модель наиболее пригодна для многих операций анализа поверхностей. С позиций КГ можно указать такие положительные черты модели: легкая процедура расчета координат каждой точки поверхности, нормали; небольшой объем информации для описания достаточно сложных форм.

К недостаткам относятся следующие: сложные формулы описания с использованием функций, которые медленно вычисляются на компьютере, снижают скорость выполнения операций отображения; невозможность в большинстве случаев применения данной формы описания непосредственно для построения изображения поверхности. В таких случаях поверхность отображают как многогранник, используя формулы аналитического описания для расчета ко-

ординат вершин граней в процессе отображения, что уменьшает скорс сравнительно с полигональной моделью описания.

Векторная полигональная модель

Для описания пространственных объектов здесь используются такие элем ты: *вершины*; отрезки прямых (*векторы*); *полилинии, полигоны*; *полигона ные поверхности* (рис. 4.2).

Элемент "вершина" (vertex) — главный элемент описания, все другие яв ются производными. При использовании трехмерной декартовой систе координаты вершин определяются как (x_i, y_i, z_i). Каждый объект однознач определяется координатами собственных вершин.

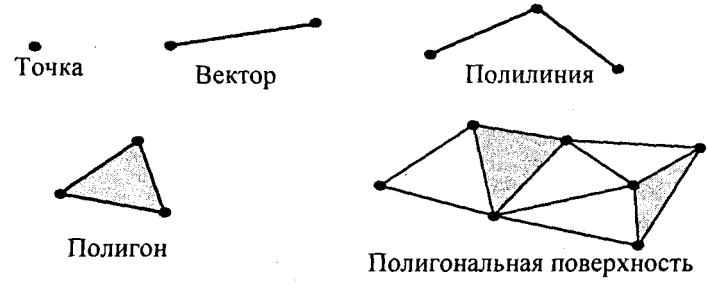


Рис. 4.2. Базовые элементы векторно-полигональной модели

Вершина может моделировать отдельный точечный объект, размер которог не имеет значения, а также может использоваться в качестве конечных точе для линейных объектов и полигонов. Двумя вершинами задается вектор. Не сколько векторов составляют полилинию. Полилиния может моделироват отдельный линейный объект, толщина которого не учитывается, а также мо жет представлять контур полигона. Полигон моделирует площадный объект Один полигон может описывать плоскую грань объемного объекта. Несколь ко граней составляют объемный объект в виде полигональной поверхно сти — многогранник или незамкнутую поверхность (в литературе часто употребляется название "полигональная сетка").

Векторную полигональную модель можно считать наиболее распространен ной в современных системах трехмерной КГ. Ее используют в системах ав томатизированного проектирования, в компьютерных играх и тренажерах, в САПР, геоинформационных системах и тому подобное.

Обсудим структуры данных, которые используются в векторной полигональ ной модели. В качестве примера объекта будет куб. Рассмотрим, как можно организовать описание такого объекта в структурах данных.

Первый способ. Сохраняем все грани в отдельности (рис. 4.3).

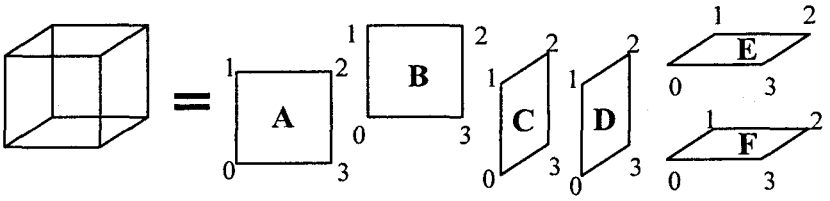


Рис. 4.3. Первый способ описания куба

Грань A = { (x_{A0},y_{A0},z_{A0}), (x_{A1},y_{A1},z_{A1}), (x_{A2},y_{A2},z_{A2}), (x_{A3},y_{A3},z_{A3}) }.

Грань B = { (x_{B0},y_{B0},z_{B0}), (x_{B1},y_{B1},z_{B1}), (x_{B2},y_{B2},z_{B2}), (x_{B3},y_{B3},z_{B3}) }.

Грань C = { (x_{C0},y_{C0},z_{C0}), (x_{C1},y_{C1},z_{C1}), (x_{C2},y_{C2},z_{C2}), (x_{C3},y_{C3},z_{C3}) }.

Грань D = { (x_{D0},y_{D0},z_{D0}), (x_{D1},y_{D1},z_{D1}), (x_{D2},y_{D2},z_{D2}), (x_{D3},y_{D3},z_{D3}) }.

Грань E = { (x_{E0},y_{E0},z_{E0}), (x_{E1},y_{E1},z_{E1}), (x_{E2},y_{E2},z_{E2}), (x_{E3},y_{E3},z_{E3}) }.

Грань F = { (x_{F0},y_{F0},z_{F0}), (x_{F1},y_{F1},z_{F1}), (x_{F2},y_{F2},z_{F2}), (x_{F3},y_{F3},z_{F3}) }.

Схематично это изобразим на рис. 4.4.

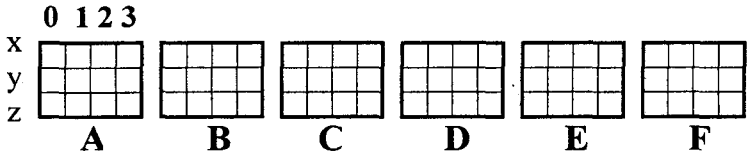


Рис. 4.4. Отдельные грани

В компьютерной программе такой способ описания объекта можно реализовать разнообразно. Например, для каждой грани открыть в памяти отдельный массив. Можно все грани записывать в один массив-вектор. А можно использовать классы (языком C++) как для описания отдельных граней, так и объектов в целом. Можно создавать структуры, которые объединяют тройки (*x, y, z*), или сохранять координаты отдельно. В значительной мере это относится уже к компетенции программиста, зависит от его вкуса. Принципиально это мало что изменяет — так или иначе в памяти необходимо сохранять координаты вершин граней плюс некоторую информацию в качестве накладных затрат.

Рассчитаем объем памяти, необходимый для описания куба следующим образом:

$$П_1 = 6 \times 4 \times 3 \times P_в,$$

где $P_в$ — разрядность чисел, необходимая для представления координат.

Шесть граней здесь описываются 24 вершинами. Такое представление избыточно — каждая вершина записана трижды. Не учитывается то, что у граней есть общие вершины.

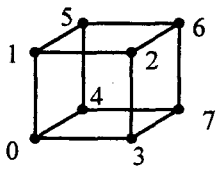


Рис. 4.5. Номера вершин

Второй способ описания. Для такого варианта координаты восьми вершин сохраняются без повторов. Вершины пронумерованы (рис. 4.5), а каждая грань дается в виде списка индексов вершин (указателей на вершины).

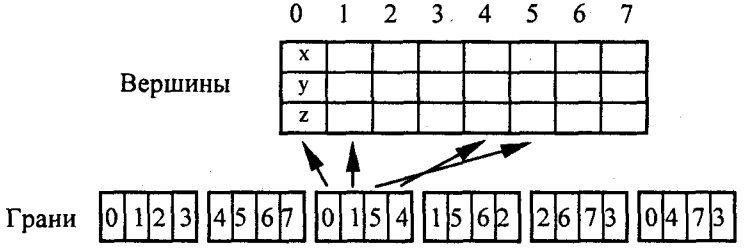


Рис. 4.6. В массивах граней сохраняются индексы вершин

Оценим затраты памяти:

$$П_2 = 8 \times 3 \times P_в + 6 \times 4 \times P_{индекс},$$

где $P_в$ — разрядность координат вершин, $P_{индекс}$ — разрядность индексов.

Третий способ описания (рис. 4.7). Этот способ (в литературе его иногда называют *линейно-узловой моделью*) основывается на иерархии: вершины-ребра-грани.

Оценим затраты памяти:

$$П_3 = 8 \times 3 \times P_в + 12 \times 2 \times P_{инд.\ вершин} + 6 \times 4 \times P_{инд.\ ребер},$$

где $P_в$ — разрядность координат, $P_{инд.\ вершин}$ и $P_{инд.\ ребер}$ — разрядность индексов вершин и ребер соответственно.

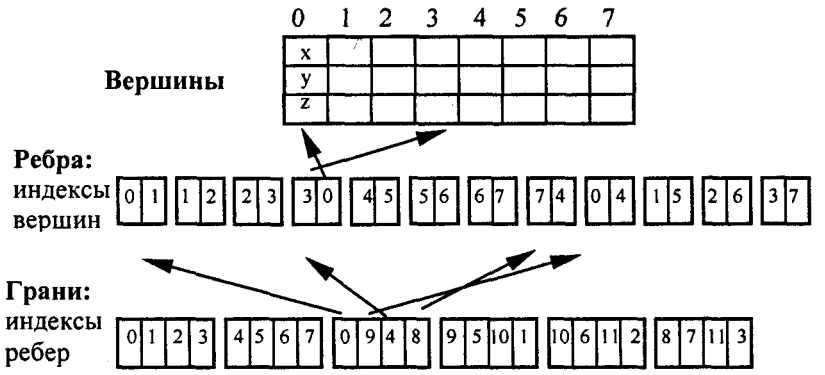


Рис. 4.7. Линейно-узловая модель

Для сравнения объемов памяти этих трех вариантов необходимо определиться с разрядностью данных. Предположим, что разрядность координат и индексов составляет четыре байта. Это соответствует, например, типу чисел с плавающей точкой **float** для координат и целому типу **long** для индексов (названия этих типов на компьютерном языке C, C++). Тогда затраты памяти в байтах составляют:

$$П_1 = 6 \times 4 \times 3 \times 4 = 288,$$
$$П_2 = 8 \times 3 \times 4 + 6 \times 4 \times 4 = 192,$$
$$П_3 = 8 \times 3 \times 4 + 12 \times 2 \times 4 + 6 \times 4 \times 4 = 288.$$

Пусть для координат отведено 8 байтов (тип с плавающей точкой **double**), а для индексов — 4 байта. Тогда:

$$П_1 = 6 \times 4 \times 3 \times 8 = 576,$$
$$П_2 = 8 \times 3 \times 8 + 6 \times 4 \times 4 = 288,$$
$$П_3 = 8 \times 3 \times 8 + 12 \times 2 \times 4 + 6 \times 4 \times 4 = 384.$$

Когда разрядность для координат больше, чем для индексов, то ощутимо преимущество второго и третьего вариантов. Наиболее экономичным можно считать второй вариант. Необходимо заметить, что такой вывод мы сделали для куба. Для других типов объектов соотношение вариантов может быть иным. Кроме того, необходимо учитывать такие варианты построения структур данных: использован ли единый массив для всех объектов, или же для каждого объекта предназначен отдельный массив (при объектно-ориентированном стиле программирования каждый объект можно сохранять в от-

дельном классе). Это может обуславливать разную необходимую разрядность для индексов.

А теперь сравним эти три разновидности векторной полигональной модели, учитывая другие аспекты.

Скорость вывода полигонов. Если для полигонов необходимо рисовать линию контура и точки заполнения, то первый и второй варианты близки по быстродействию — и контуры, и заполнения рисуются одинаково. Отличия в том, что для второго варианта сначала надо выбирать индекс вершины, что замедляет процесс вывода. В обоих случаях для смежных граней повторно рисуется общая часть контура. Для третьего варианта можно предусмотреть более совершенный способ рисования контура — каждая линия будет рисоваться только один раз, если в массивах описания ребер предусмотреть бит, означающий, что это ребро уже нарисовано. Это обуславливает преимущества третьего варианта по быстродействию.

Блокирование повторного рисования линий контуров смежных граней позволяет решить также проблему искажения стиля линий, если линии контуров не сплошные, а, например, пунктирные.

Топологический аспект. Представим, что имеется несколько смежных граней. Что будет, если изменить координаты одной вершины в структурах данных? Результат приведен на рис. 4.8.

Рис. 4.8. Результат изменения координат одной вершины

Поскольку для второго и третьего вариантов каждая вершина сохраняется в одном экземпляре, то изменение ее координат автоматически приводит к изменению всех граней, в описании которых сохраняется индекс этой вершины. Это полезно, например, в геоинформационных системах при описании соседних земельных участков или других смежных объектов.

Следует заметить, что подобного результата можно достичь и при структуре данных, соответствующей первому варианту. Можно предусмотреть поиск других вершин, координаты которых совпадают с координатами точки *A*.

Иначе говоря, поддержка такой операции может быть обеспечена как структурами данных, так и алгоритмически.

Однако когда нужно разъединить смежные грани, то для второго и третьего вариантов это сложнее, чем для первого — необходимо записать в массивы новую вершину, новые ребра и определить индексы в массивах граней.

При разработке новой графической системы обычно приходится решать такой вопрос: какие операции реализовывать только алгоритмически, а какие обеспечивать структурами данных? Ответ на это можно дать, проанализировав много других факторов. Здесь мы рассмотрели только малую часть из них.

Положительные черты векторной полигональной модели:

- удобство масштабирования объектов. При увеличении или уменьшении объекты выглядят более качественно, чем при растровых моделях описания. Диапазон масштабирования определяется точностью аппроксимации и разрядностью чисел для представления координат вершин;
- небольшой объем данных для описания простых поверхностей, которые адекватно аппроксимируются плоскими гранями;
- необходимость вычислять только координаты вершин при преобразованиях систем координат или перемещении объектов;
- аппаратная поддержка многих операций в современных графических видеосистемах, которая обуславливает достаточную скорость для анимации.

Недостатки полигональной модели:

- сложные алгоритмы визуализации для создания реалистичных изображений; сложные алгоритмы выполнения топологических операций, таких, например, как разрезы;
- аппроксимация плоскими гранями приводит к погрешности моделирования. При моделировании поверхностей, которые имеют сложную фрактальную форму, обычно невозможно увеличивать количество граней из-за ограничений по быстродействию и объему памяти компьютера.

Воксельная модель

Воксельная модель — **это трехмерный растр**. Подобно тому, как пикселы располагаются на плоскости 2D-изображения, так и вокселы образовывают трехмерные объекты в определенном объеме (рис. 4.9). Воксел — это элемент объема (**voxel** — volume element).

Как мы знаем, каждый пиксел должен иметь свой цвет. Каждый воксел также имеет свой цвет, а, кроме того, прозрачность. Полная прозрачность воксела

означает пустоту соответствующей точки объема. При моделировании объема каждый воксел представляет элемент объема определенного размера. Чем больше вокселов в определенном объеме и меньше размер вокселов, тем точнее моделируются трехмерные объекты — увеличивается разрешающая способность.

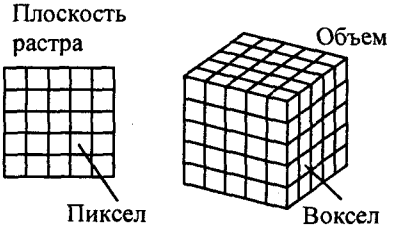


Рис. 4.9. Пикселы и вокселы

Для современной КГ воксельный метод считается одним из перспективных. Его используют в компьютерных системах для медицины. Например, при сканировании томографом (computer tomography) получаются изображения срезов объекта, которые потом объединяются в виде объемной модели для дальнейшего анализа [53]. Воксельный метод используется в геологии, сейсмологии, в компьютерных играх [50, 56]. Вокселы также используются для графических устройств отображения, которые создают действительно объемные изображения [37].

Положительные черты воксельной модели:

- позволяет достаточно просто описывать сложные объекты и сцены; простая процедура отображения объемных сцен;
- простое выполнение топологических операций над отдельными объектами и сценой в целом. Например, просто выполняется показ разреза — для этого соответствующие вокселы можно сделать прозрачными.

Недостатки воксельной модели:

- большое количество информации, необходимой для представления объемных данных. Например, объем 256×256×256 имеет небольшую разрешающую способность, но требует свыше 16 миллионов вокселов;
- значительные затраты памяти ограничивают разрешающую способность, точность моделирования; большое количество вокселов обусловливает малую скорость создания изображений объемных сцен;
- как и для любого растра, возникают проблемы при увеличении или уменьшении изображения. Например, при увеличении ухудшается разрешающая способность изображения.

Равномерная сетка

Эта модель описывает координаты отдельных точек поверхности следующим способом (рис. 4.10). Каждому узлу сетки с индексами (*i, j*) приписывается значение высоты z_{ij}. Индексам (*i, j*) отвечают определенные значения координат (*x, y*). Расстояние между узлами одинаковое — *dx* по оси *x* и *dy* по оси *y*.

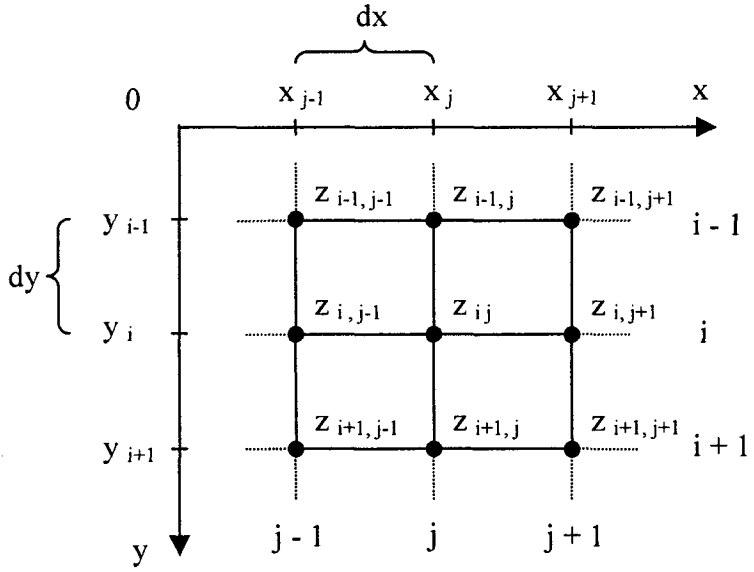


Рис. 4.10. Узлы равномерной сетки

Фактически, такая модель — двумерный массив, растр, матрица, каждый элемент которой сохраняет значение высоты.

Не каждая поверхность может быть представлена этой моделью. Если в каждом узле записывается только одно значение высоты, то это означает, что поверхность описывается однозначной функцией $z = f(x, y)$. Иначе говоря, это такая поверхность, которую любая вертикаль пересекает только один раз. Не могут моделироваться также вертикальные грани. Необходимо заметить, что для сетки не обязательно использовать только декартовые координаты. Например, для того чтобы описать поверхность шара однозначной функцией, можно использовать полярные координаты. Равномерная сетка часто используется для описания рельефа земной поверхности.

Рассмотрим, как можно вычислить значения высоты для любой точки внутри границ сетки. Пусть ее координаты равны (*x, y*). Надо найти соответствую-

щее значение z. Решением такой задачи является интерполяция значений координат z ближайших узлов (рис. 4.11).

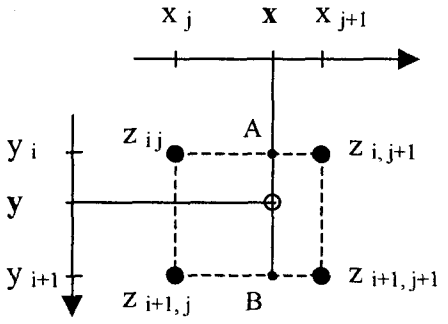


Рис. 4.11. Точка в сетке с координатами (x, y, z)

Сначала необходимо вычислить индексы j и i одного из узлов:

$$j = \left]\frac{x - x_0}{dx}\right[,$$

$$i = \left]\frac{y - y_0}{dy}\right[,$$

где $]a[$ — целая часть числа a, то есть наибольшее целое, которое не превышает a.

Далее используем, например, линейную интерполяцию. Для этого сначала найдем значения z в точках A и B. Из пропорции

$$\frac{z_A - z_{i,j}}{z_{i,j+1} - z_{i,j}} = \frac{x - x_j}{x_{j+1} - x_j},$$

учитывая, что $x_{j+1} - x_j = dx$, получим $z_A = z_{ij} + (x - x_j)(z_{i,j+1} - z_{ij})/dx$.

Аналогично найдем z_B: $z_B = z_{i+1,j} + (x - x_j)(z_{i+1,j+1} - z_{i+1,j})/dx$.

Теперь можно найти нужное значение z, поделив отрезок AB пропорционально значению y:

$$\frac{z - z_A}{z_B - z_A} = \frac{y - y_i}{dy}.$$

Получим $z = z_A + (y - y_i)(z_B - z_A)/dy$.

Положительные черты равномерной сетки:

- простота описания поверхностей;
- возможность быстро узнать высоту любой точки поверхности простой интерполяцией.

Недостатки равномерной сетки:

- поверхности, которые соответствуют неоднозначной функции высоты в узлах сетки, не могут моделироваться;
- для описания сложных поверхностей необходимо большое количество узлов, которое может быть ограничено объемом памяти компьютера;
- описание отдельных типов поверхностей может быть сложнее, чем в других моделях. Например, многогранная поверхность требует избыточный объем данных для описания по сравнению с полигональной моделью.

Неравномерная сетка. Изолинии

Неравномерной сеткой назовем модель описания поверхности в виде множества отдельных точек $\{(x_0, y_0, z_0), (x_1, y_1, z_1), ..., (x_{n-1}, y_{n-1}, z_{n-1})\}$, принадлежащих поверхности. Эти точки могут быть получены, например, в результате измерений поверхности какого-нибудь объекта с помощью определенного оборудования.

Такую модель можно считать обобщением для некоторых рассмотренных нами моделей. Например, векторная полигональная модель и равномерная сетка могут считаться разновидностями неравномерной сетки. Эти разновидности мы рассмотрели в отдельности, так как они играют важную роль для решения задач КГ. А вообще, может существовать много вариантов классификации способов описания поверхностей. Следует учитывать определенную условность нашего перечня моделей поверхностей, последовательность перечисления таких моделей может быть и другой.

Рассмотрим модель поверхности в виде ***множества точечных значений***, логически никак не связанных между собой. Неравномерность задания опорных точек усложняет определение координат для других точек поверхности, которые не совпадают с опорными точками. Нужны специальные методы пространственной интерполяции. Так, например, можно поставить такую задачу — по известным координатам (x, y) вычислить значения координаты z. Для этого необходимо найти несколько самых близких точек, а потом вычислить искомое значение z, исходя из взаимного расположения этих точек в проекции (x, y). Как мы уже рассмотрели выше, для равномерной сетки это намного проще — поиска фактически нет, мы сразу рассчитываем индексы самых близких опорных точек. Еще одна задача — отобразить поверхность.

Эту задачу можно решать несколькими способами, в том числе ***триангуляцией***. Процесс триангуляции можно представить себе так (рис. 4.12). Сначала находим первые три самые близкие друг другу точки — и получаем одну плоскую треугольную грань. Потом находим точку, ближайшую к этой грани, и образовываем смежную грань. И так далее, пока не останется ни одной отдельной точки. Это общая схема, в литературе описано много разных способов триангуляции. Довольно часты ссылки на триангуляцию Делоне [48].

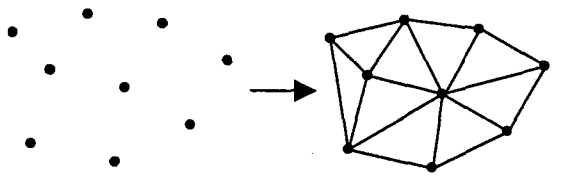


Рис. 4.12. Триангуляция неравномерной сетки

Описание поверхности треугольными гранями можно уже считать разновидностью векторной полигональной модели. В англоязычной литературе для нее встречается такое название: **TIN** (Triangulated Irregular Network). После триангуляции получаем полигональную поверхность, отображение которой сделать уже достаточно просто.

Рассмотрим еще один из вариантов описания поверхности — ***изолинии высоты***. Любая изолиния состоит из точек, представляющих одно числовое значение какого-то показателя, в данном случае — значение высоты (рис. 4.13, 4.14). Изолинии высоты также можно вообразить себе как контуры разреза поверхности горизонтальными плоскостями (поэтому для изолиний высоты часто применяется название "горизонтали").

Описание поверхности изолиниями высоты часто используется, например, в картографии. По бумажной карте можно с определенной точностью рассчитать высоты в точках местности, углы наклона и прочие параметры рельефа. Необходимо заметить, что описание рельефа земной поверхности изолиниями высоты неправильно представлять как разрезы горизонтальными плоскостями, ибо поверхность Земли не плоская. Если бы Земля была шаром, то изолинии высоты можно было бы трактовать как изолинии радиусов. Однако Земля — это не шар, она имеет намного более сложную форму, названную *геоидом*. В геодезии и картографии геоид аппроксимируют с определенной точностью разнообразными эллипсоидами. Таким образом, здесь можно говорить об изолиниях некоторых условных высот в специальных системах координат.

Конечно, для описания поверхности можно использовать не только изолинии высоты, но и другие изолинии, например x- или y-изолинии.

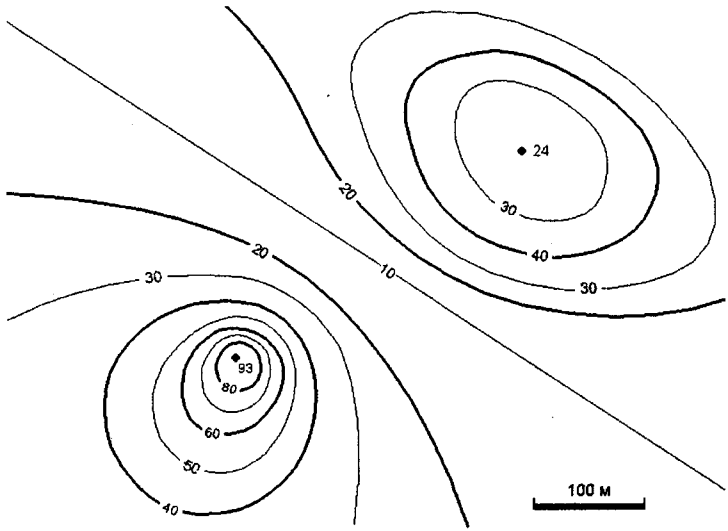


Рис. 4.13. Поверхность задана изолиниями и отметками высоты

Рис. 4.14. Та же поверхность в аксонометрической проекции

В компьютерных системах изолинии часто описываются векторно — полилиниями. Используются также изолинии в виде сплайновых кривых.

Точки, которые составляют изолинии, и отдельные опорные точки располагаются неравномерно. Это усложняет расчет координат точек поверхности. В графических компьютерных системах для выполнения многих операций, и в первую очередь — для показа поверхности, обычно необходимо преобразовывать описание поверхности в другую форму. Преобразование изолиний в полигональную модель также выполняется методами триангуляции (здесь алгоритмы триангуляции сложнее, чем для триангуляции отдельных точек). Для преобразования неравномерной сетки в равномерную используют специальную интерполяцию.

Положительные черты неравномерной сетки: использование отдельных опорных точек, наиболее важных для заданной формы поверхности, обу-

славливает меньший объем информации по сравнению с другими моделями, например, с равномерной сеткой; наглядность показа рельефа поверхности изолиниями на картах, планах.

Недостатки: невозможность или сложность выполнения многих операций над поверхностями; сложные алгоритмы преобразования в другие формы описания поверхностей.

Преобразование моделей описания поверхности

Рассмотрим, как можно выполнить преобразование моделей описания поверхности. Сделаем это на примере преобразования неравномерной сетки в равномерную. Задачу можно сформулировать так: поверхность описана в виде точечных значений, изолиний и площадных изообластей. Необходимо построить равномерную сетку так, чтобы она представляла эту поверхность с определенной точностью.

Для решения данной задачи можно использовать алгоритм, предложенный автором этой книги и реализованный в геоинформационной системе ГИС "ОКО" в 1996 году [57].

Сначала рассмотрим аспекты точности алгоритма и ограничения для его использования.

Равномерную сетку можно рассматривать как растр. Расстояние между узлами сетки в плоскости ($x0y$) обуславливает разрешающую способность такого растра и определяет точность моделирования по осям x и y. Конечно, чем меньше расстояние между узлами, тем больше точность моделирования, но это ведет к возрастанию количества узлов и соответственно увеличивает размеры растра. Таким образом, мы определили размер растра по горизонтали (c_x) и вертикали (c_y).

Необходимо также учесть дискретность представления чисел в компьютере при хранении в памяти значений в узлах сетки. В современных цифровых компьютерах числа обычно представляются в форматах с разрядностью, кратной 8 (байт). Однобайтовые целые числа дают 256 градаций, двухбайтовые — 65 536 и так далее. Можно также использовать и форматы с плавающей точкой.

Выбираем формат чисел для кодирования пикселов растра. Одной из основных особенностей предложенного алгоритма является то, что число 0 для каждого пиксела указывает на неопределенное значение высоты (пустоты до интерполяции). Это означает, что, например, для однобайтовых пикселов высота имеет не 256, а 255 градаций. *Дискретность значений высоты = = диапазон значений / 255.*

Разумеется, лучше использовать бóльшую разрядность, чем 8 бит. Основное ограничение — объем памяти, необходимый для растра:

$П = cx \cdot cy \cdot$ (байтов на пиксел).

Все это необходимо учитывать для построения равномерной сетки поверхности с необходимой точностью.

Общая схема алгоритма

1. Открываются два массива для растров: *A* и *B*. Каждый растр имеет размер $cx \times cy$ пикселов.
2. В растре *A* обнуляются все его пикселы.
3. В растре *A* отображаются изообласти высот в виде точечных значений, изолиний и заполненных фигур (полигонов). Для отображения этих элементов используем известный алгоритм растеризации для линий и полигонов (для точек это тривиально — отдельные пикселы). Каждый пиксел рабочего растра представляет значение высоты согласно выбранному способу кодирования. Нулевые пикселы растра (пустоты) отвечают неопределенности высоты.
4. Заполняются пустоты. В процессе заполнения пикселы результата записываются в растр *B*. В ходе заполнения пустот вычисляется R_{max}.
5. Если $R_{max} < 2$, то интерполяция закончена. Результат — равномерная сетка — находится в массивах растра *B*.
6. Проводятся контуры на границах раздела областей заполнения. В процессе оконтуривания пикселы результата записываются в растр *A*.
7. Переход к пункту 4.

В результате работы этого алгоритма мы получаем растр, в котором нет ни одного нулевого пиксела, то есть высоты определены для всех узлов сетки.

Продолжим описание алгоритма согласно приведенной выше общей схеме. В пункте 3 точечные, линейные и площадные изообласти высоты отображаются как обычные пикселы, линии и полигоны. При таком отображении существует проблема "одного пиксела". Она обусловлена тем, что некоторые алгоритмы растеризации могут различно располагать пикселы, например, для линии. Для линии необходимо использовать алгоритмы, способные отрабатывать нецелые координаты вершин. Это касается и полигонов. Конечно, использование таких алгоритмов приводит к уменьшению скорости, но главное здесь — точность. Точность растеризации в значительной степени предопределяет точность преобразования в равномерную сетку.

Заполнение пустот выполняется следующим образом:

```
1. Rmax = 0;
2.  for (y = 0; y < cy; y++)
3.   for (x = 0; x < cx; x++)
4.   Если пиксел (x,y) растра A ненулевой, то копируем в растр B.
     Иначе:
     {
5.   Поиск ближайшего ненулевого пиксела. В результате поиска
становятся известными значение пиксела (цвет) и его расстояние (r)
до точки (x,y).
6.   Если ненулевой пиксел не найден, то прекращаем весь процесс
     и делаем сообщения об ошибке.
7.   Записываем в растр B по координатам (x,y) цвет найденного пиксела.
8.   Если R max < r, то R max = r
     }
```

В ходе заполнения анализируются пикселы рабочего растра, и результаты записываются в другой растр. Для этого и предусмотрены два массива. Необходимо отметить, что пункт 4 алгоритма заполнения можно упростить, если при его выполнении не копировать ненулевые пикселы, а перед началом сканирования растра (п. 2) сразу скопировать весь рабочий растр в другой массив. Это позволяет ускорить работу при условии, что групповое копирование блоков памяти выполняется быстрее.

Проведение контуров. Оконтуривание можно выполнить методами локальной фильтрации изображения растра. Например, таким способом:

```
1. for (y = 0; y < cy; y++)
2.  for (x = 0; x < cx; x++)
3.    Если пиксел (x, y) растра A ненулевой, то:
       {
4.      Если в растре B пиксел (x, y) не равен пикселу (x+1, y),
           или пиксел (x, y) не равен пикселу (x, y+1),
        то:
           {
5.         Если пиксел (x, y) равен пикселу (x+1, y),
           то в растр A записывается значение
                  (пиксел (x, y) + пиксел (x, y+1)) / 2
7.          иначе : значение (пиксел (x, y) + пиксел (x+1, y)) / 2
           }
       }
```

В процессе оконтуривания в растре *A* появляются новые контуры на границе областей заполнения. Линии контуров образовываются пикселами, значение

(цвет) которых равно полусумме пикселов областей заполнения, а располагаются новые контуры посредине старых, то есть, выполняется линейная интерполяция высоты. В растре *A* также хранятся предыдущие линии контуров. Таким образом, с каждым циклом заполнения-оконтуривания количество контуров изолинии удваивается. Так длится до тех пор, пока контурные линии не сомкнутся — нечего будет заполнять. Количество циклов интерполяции оценивается как двоичный логарифм расстояния между ненулевыми пикселами растра, на котором отображены исходные данные.

Поиск самого близкого ненулевого пиксела. Эта процедура использована в алгоритме заполнения. Для того чтобы найти пиксел, ближайший к точке (x,y), можно, казалось бы, сделать, так: последовательно увеличивать радиус, анализируя пикселы, располагающиеся на окружности. Однако в растре это делать нельзя. Если увеличивать радиус окружности, используя +1, то будет пропущено много точек растра, а если радиус окружности увеличивать шагами, меньшими 1, то много пикселов будет проанализировано повторно.

Быстро и просто этот поиск можно осуществить, если идти по контуру квадрата (рис. 4.15). Однако точки на периметре квадрата имеют различное расстояние до центра. Решение этой проблемы заключается в организации двухэтапного цикла поиска. Сначала последовательно увеличиваем размер квадрата с центром в точке (x, y) до тех пор, пока на периметре не обнаружится ненулевой (значащий) пиксел растра. Рассчитываем расстояние R_1. Если это расстояние больше, чем (размер квадрата + 1), то начинаем второй этап поиска. Для этого продолжаем увеличивать размер квадрата вплоть до значения R_1. Если при этом находятся новые точки с расстоянием $R_2 < R_1$, то поиск продолжается, но максимальный размер окрестности ограничиваем уже R_2. Здесь необходимо пояснить, что в качестве размера квадрата здесь считается половина его стороны.

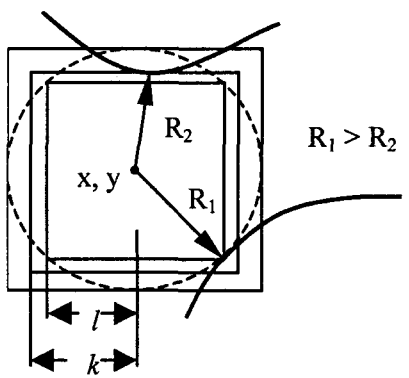


Рис. 4.15. Поиск по периметру квадратов

Дадим запись алгоритма поиска ближайшей точки:

```
1. l = 1; c = 0; r = максимальное целое число ;
2. Поиск по периметру квадрата ППК(l, x, y, r)  для l=1, 2, 3, ...
3. Если найден пиксел, то:
       {
4.     Расстояние r = R1; значение пиксела c = c1 ;
5.     Если r > l + 1, то:
           {
6.          k = l + 1 ; kmax = r ;
7.          Поиск по периметру квадрата ППК(k, x, y, r) ;
8.          Если найден пиксел и R2 < r, то r = R2 ;  c = c2 ;
9.          k = k + 1 ;
10.         Если k < kmax , то перейти к пункту 7.
           }
       }
11. Если c не нуль, то это означает, что искомая точка найдена.
Расстояние до этой точки (r) и цвет пиксела используются далее
в алгоритме заполнения.
```

Процедура поиска пикселов по периметру квадрата названа здесь ППК(l, x_c, y_c, r), где l — половинный размер квадрата; x_c и y_c — координаты центра квадрата; r — расстояние для сравнения — если находится пиксел с большим расстоянием, то он не учитывается. В результате работы процедуры ППК определяется цвет найденного пиксела (c) и рассчитывается расстояние до центра (R).

Алгоритм для ППК может быть, например, таким:

```
1. x = 0 ; y = l ; d = 1 ; R^2 = l^2; r^2max = r^2;
2. c = Пиксел (xc+x, yc+y) ; Если c не нуль, то на 14 ;
3. c = Пиксел (xc-x, yc+y) ; Если c не нуль, то на 14 ;
4. c = Пиксел (xc+x, yc-y) ; Если c не нуль, то на 14 ;
5. c = Пиксел (xc-x, yc-y) ; Если c не нуль, то на 14 ;
6. c = Пиксел (xc+y, yc+x) ; Если c не нуль, то на 14 ;
7. c = Пиксел (xc-y, yc+x) ; Если c не нуль, то на 14 ;
8. c = Пиксел (xc+y, yc-x) ; Если c не нуль, то на 14 ;
9. c = Пиксел (xc-y, yc-x) ; Если c не нуль, то на 14 ;
10. R^2 = R^2 + 1 ;
11. d = d + 2 ;
12. x = x + 1 ;
13. Если R^2 <= r^2max, то на 2.
14. R = корень квадратный из  R^2.
```

Рассмотрим пример работы приведенного алгоритма преобразования неравномерной сетки в равномерную. На рис. 4.16 изображена модель некоторой поверхности в виде изообластей высоты.

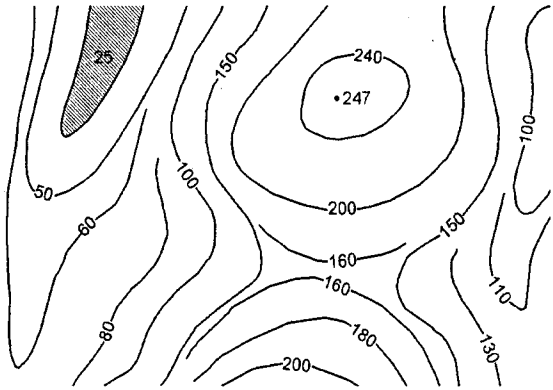


Рис. 4.16. Пример задания поверхности неравномерной сеткой

После первого заполнения растр имеет следующий вид (рис. 4.17).

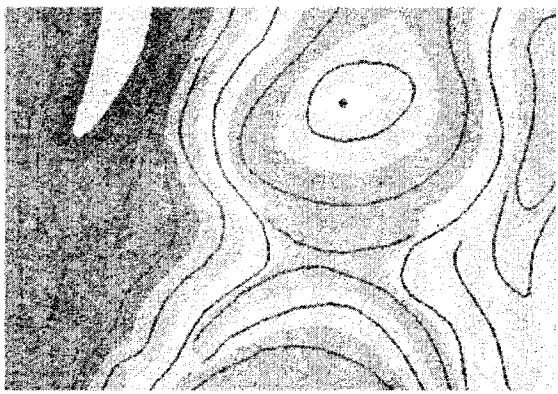


Рис. 4.17. Вид растра после первого заполнения

Это уже равномерная сетка высоты. Однако поверхность будет негладкой, иметь большие ступени. Необходимо продолжать интерполяцию дальше.

Потом выполняется оконтуривание (рис. 4.18).

И так далее — выполняются циклы заполнения-оконтуривания. Всего было выполнено 7 циклов. После выполнения каждого цикла равномерная сетка все более точно соответствует гладкой поверхности. Это изображено на рис. 4.19.

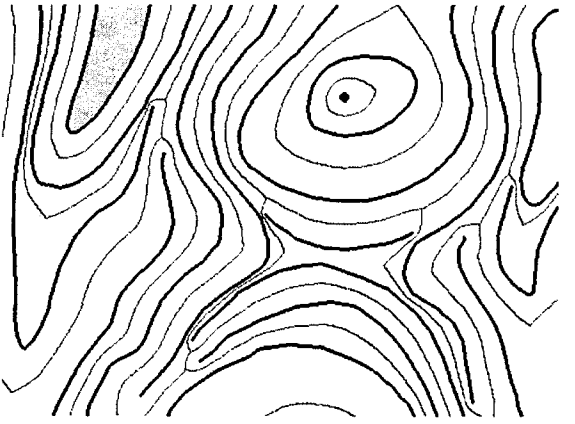
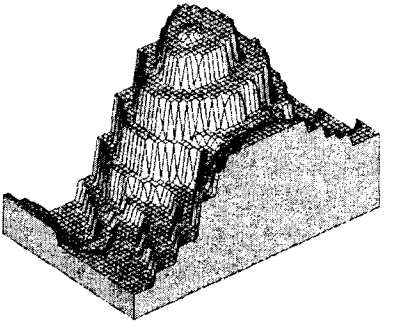
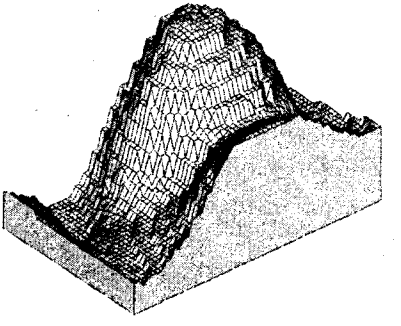


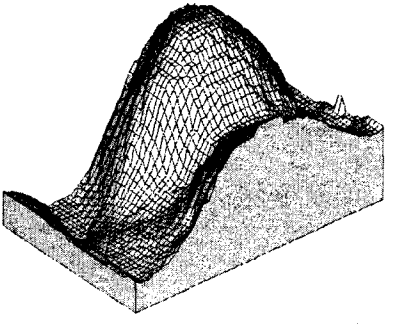
Рис. 4.18. Вид растра после оконтуривания



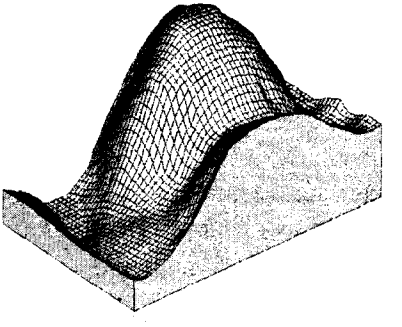
Выполнен один цикл



Два цикла



Три цикла



Семь циклов

Рис. 4.19. Результаты выполнения циклов заполнения-оконтуривания

4.2. Визуализация объемных изображений

Любой объект, в том числе и объемный, может быть изображен различными способами. В одном случае необходимо показать внутреннюю структуру объектов, в другом — внешнюю форму объекта, в третьем — имитировать реальную действительность, в четвертом — поразить воображение зрителя чем-то неизвестным. Условно разделим способы визуализации по характеру изображений и по степени сложности соответствующих алгоритмов на такие уровни:

1. Каркасная ("проволочная") модель.
2. Показ поверхностей в виде многогранников с плоскими гранями или сплайнов с удалением невидимых точек.
3. То же, что и для второго уровня, плюс сложное закрашивание объектов для имитации отражения света, затенения, прозрачности, использование текстур.

На рис. 4.20—4.26 изображены различные способы показа трехмерных объектов в порядке усложнения используемых алгоритмов.

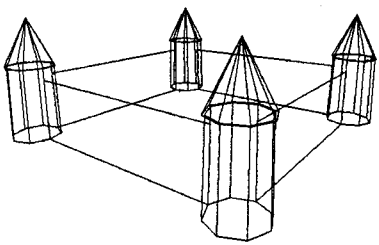


Рис. 4.20. Каркасная модель

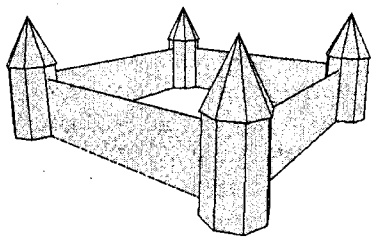


Рис. 4.21. Удаление невидимых точек

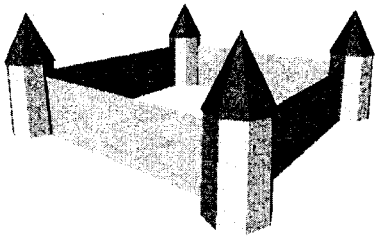


Рис. 4.22. Закрашивание граней с учетом освещения

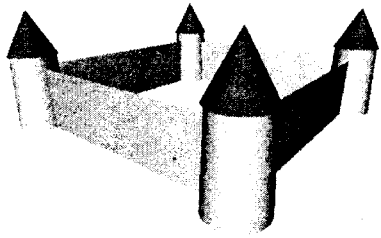


Рис. 4.23. Имитация гладких поверхностей закрашиванием

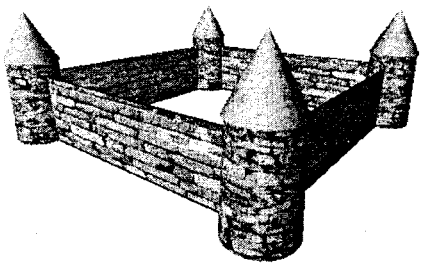


Рис. 4.24. Наложение текстуры

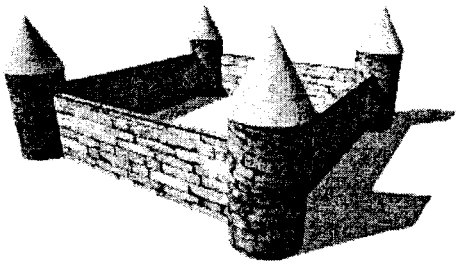


Рис. 4.25. Тени

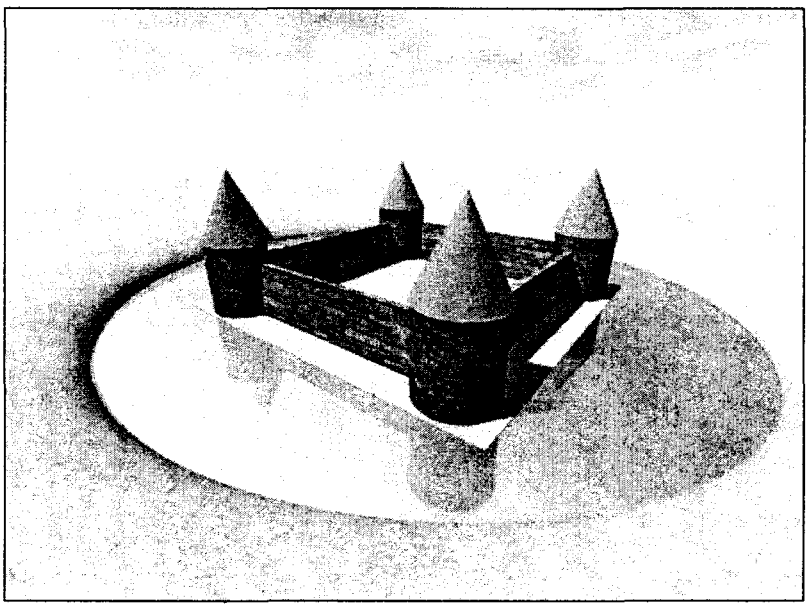


Рис. 4.26. Добавлены фон, матовые и зеркальные поверхности

Простейшая каркасная модель часто применяется в процессе редактировани объемных объектов. Визуализация второго уровня используется для упро щенного показа объемных объектов. Например, для графиков функци $z = f(x,y)$ (в виде рельефа поверхности) часто достаточно показать все гран сетки одним цветом, но зато необходимо обязательно удалить невидимы точки. Это более сложная процедура по сравнению с выводом каркасног изображения.

Сложность процесса графического вывода возрастает по мере приближения некоторому идеалу для компьютерной графики — создания полной иллюзи естественных, живых, реалистичных изображений. Усилия многих ученых

инженеров во всем мире направлены на разработку методов и средств достижения этой цели. В этом плане наиболее полно ощущается связь компьютерной графики с естественными науками, с дисциплинами, посвященными изучению окружающего нас мира. Например, для создания реалистичных изображений необходимо учитывать законы оптики, описывающие свет и тень, отражение и преломление. Компьютерная графика находится на стыке многих дисциплин и разделов науки.

Каркасная визуализация

Каркас обычно состоит из отрезков прямых линий (соответствует многограннику), хотя можно строить каркас и на основе кривых, в частности сплайновых кривых Безье. Все ребра, показанные в окне вывода, видны — как ближние, так и дальние.

Для построения каркасного изображения надо знать координаты всех вершин в мировой системе координат. Потом преобразовать координаты каждой вершины в экранные координаты в соответствии с выбранной проекцией. Затем выполнить цикл вывода в плоскости экрана всех ребер как отрезков прямых (или кривых), соединяющих вершины.

Показ с удалением невидимых точек

Здесь мы будем рассматривать поверхности в виде многогранников или полигональных сеток. Известны такие методы показа с удалением невидимых точек: сортировка граней по глубине, метод плавающего горизонта, метод Z-буфера [28, 32].

Сортировка граней по глубине. Это означает рисование полигонов граней в порядке от самых дальних к самым близким. Этот метод не является универсальным, ибо иногда нельзя четко различить, какая грань ближе (рис. 4.27). Известны модификации этого метода, которые позволяют корректно рисовать такие грани, они описаны в [28]. Метод сортировки по глубине эффективен для показа поверхностей, заданных функциями $z = f(x,y)$ (рис. 4.28).

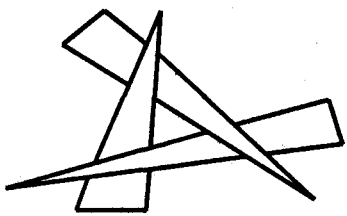
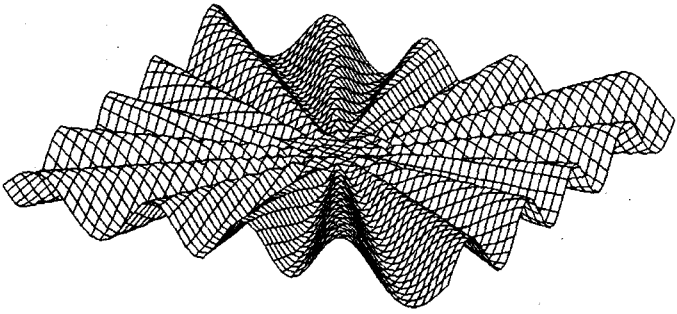


Рис. 4.27. В каком порядке рисовать эти грани?



$$z = \sqrt{x^2 + y^2}\,\sin\left(16 \times \operatorname{arctg}\left(\frac{x}{y}\right)\right).$$

Рис. 4.28. Эта поверхность нарисована четырехугольными гранями

Метод плавающего горизонта. В отличие от предыдущего метода при методе плавающего горизонта грани выводятся в последовательности от ближайших к самым дальним. На каждом шаге границы граней образовывают две ломаные линии — верхний горизонт и нижний горизонт. Во время вывода каждой новой грани рисуется только то, что выше верхнего горизонта, и то, что ниже нижнего горизонта. Соответственно, каждая новая грань поднимает верхний и опускает нижний горизонты. Этот метод часто используют для показа поверхностей, которые описываются функциями $z = f(x,y)$.

Метод Z-буфера. Метод основывается на использовании дополнительного массива, буфера в памяти, в котором сохраняются координаты Z для каждого пиксела растра. Координата Z отвечает расстоянию точек пространственных объектов до плоскости проецирования. Например, она может быть экранной координатой Z в системе экранных координат (X,Y,Z), если Z перпендикулярна плоскости экрана.

Рассмотрим алгоритм рисования объектов согласно этому методу. Пусть, чем ближе точка в пространстве к плоскости проецирования, тем больше значение Z. Тогда сначала Z-буфер заполняется минимальными значениями. Потом начинается вывод всех объектов. Причем не имеет значение порядок вывода объектов. Для каждого объекта выводятся все его пикселы в любом порядке. Во время вывода каждого пиксела по его координатам (X,Y) находится текущее значение Z в Z-буфере. Если рисуемый пиксел имеет большее значение Z, чем значение в Z-буфере, то это означает, что эта точка ближе к объекту. В этом случае пиксел действительно рисуется, а его Z-координата записывается в Z-буфер. Таким образом, после рисования всех пикселов всех объектов растровое изображение будет состоять из пикселов, которые соот-

ветствуют точкам объектов с самыми большими значениями координат Z, то есть видимые точки — ближе всех к нам.

Этот метод прост и эффективен благодаря тому, что не нужно ни сортировать объекты, ни сортировать их точки. При рисовании объектов, которые описываются многогранниками или полигональными сетками, манипуляции со значениями Z-буфера легко совместить с выводом пикселов заполнения полигонов плоских граней.

В настоящее время метод Z-буфера используется во многих графических 3d-акселераторах, которые аппаратно реализуют этот метод. Наиболее целесообразно, когда акселератор имеет собственную память для Z-буфера, доступ к которой осуществляется быстрее, чем к оперативной памяти компьютера. Возможности аппаратной реализации используются разработчиками и пользователями компьютерной анимации, позволяя достичь большой скорости прорисовки кадров.

4.3. Закрашивание поверхностей

В этом разделе мы рассмотрим методы, которые позволяют получить более-менее реалистичные изображения для объектов, моделируемых многогранниками и полигональными сетками. Эти методы достаточно подробно описаны в [9, 28, 32], а также в [58, 60].

Модели отражения света

Рассмотрим, как можно определить цвет пикселов изображения поверхности согласно интенсивности отраженного света при учете взаимного расположения поверхности, источника света и наблюдателя.

Зеркальное отражение света. Угол между нормалью и падающим лучом (θ) равен углу между нормалью и отраженным лучом. Падающий луч, отраженный, и нормаль располагаются в одной плоскости (рис. 4.29).

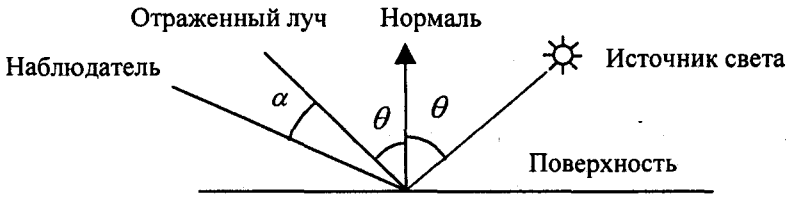


Рис. 4.29. Зеркальное отражение света

Поверхность считается *идеально зеркальной*, если на ней отсутствуют какие-либо неровности, шероховатости. Собственный цвет у такой поверхности не наблюдается. Световая энергия падающего луча отражается только по линии отраженного луча. Какое-либо рассеяние в стороны от этой линии отсутствует. В природе, вероятно, нет идеально гладких поверхностей, поэтому полагают, что если глубина шероховатостей существенно меньше длины волны излучения, то рассеивания не наблюдается. Для видимого спектра можно принять, что глубина шероховатостей поверхности зеркала должна быть существенно меньше 0.5 мкм [30].

Если поверхность зеркала отполирована неидеально, то наблюдается зависимость интенсивности отраженного света от длины волны — чем больше длина волны, тем лучше отражение. Например, красные лучи отражаются сильнее, чем синие.

При наличии шероховатостей имеется зависимость интенсивности отраженного света от угла падения. Отражение света максимально для углов θ, близких к 90 градусам [13, 30].

Падающий луч, попадая на слегка шероховатую поверхность реального зеркала, порождает не один отраженный луч, а несколько лучей, рассеиваемых по различным направлениям. Зона рассеивания зависит от качества полировки и может быть описана некоторым законом распределения. Как правило, форма зоны рассеивания симметрична относительно линии идеального зеркально отраженного луча. К числу простейших, но достаточно часто используемых, относится эмпирическая модель распределения Фонга, согласно которой интенсивность зеркально отраженного излучения пропорциональна $(\cos\alpha)^p$, где α — угол отклонения от линии идеально отраженного луча. Показатель p находится в диапазоне от 1 до 200 и зависит от качества полировки [28]. Запишем это таким образом:

$$I_s = I K_s \cos^p \alpha,$$

где I — интенсивность излучения источника, K_s — коэффициент пропорциональности.

Диффузное отражение. Этот вид отражения присущ *матовым* поверхностям. Матовой можно считать такую поверхность, размер шероховатостей которой уже настолько велик, что падающий луч рассеивается равномерно во все стороны. Такой тип отражения характерен, например, для гипса, песка, бумаги. Диффузное отражение описывается законом Ламберта, согласно которому интенсивность отраженного света пропорциональна косинусу угла

между направлением на точечный источник света и нормалью к поверхности (рис. 4.30).

$$I_d = I K_d \cos \theta,$$

где I — интенсивность источника света, K_d — коэффициент, который учитывает свойства материала поверхности. Значение K_d находится в диапазоне от 0 до 1 [28]. Интенсивность отраженного света не зависит от расположения наблюдателя.

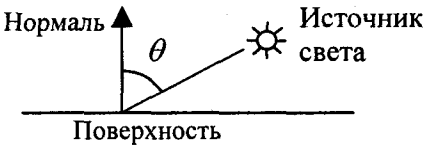


Рис. 4.30. Матовая поверхность

Матовая поверхность имеет свой цвет. Наблюдаемый цвет матовой поверхности определяется комбинацией собственного цвета поверхности и цвета излучения источника света.

При создании реалистичных изображений следует учитывать то, что в природе, вероятно, не существует идеально зеркальных или полностью матовых поверхностей. При изображении объектов средствами компьютерной графики обычно моделируют сочетание зеркальности и диффузного рассеивания в пропорции, характерной для конкретного материала. В этом случае модель отражения записывают в виде суммы диффузной и зеркальной компонент:

$$I_{отр} = I (K_d \cos \theta + K_s \cos^p \alpha),$$

где константы K_d, K_s определяют отражательные свойства материала.

Согласно этой формуле интенсивность отраженного света равна нулю для некоторых углов θ и α. Однако в реальных сценах обычно нет полностью затемненных объектов, следует учитывать фоновую подсветку, освещение рассеянным светом, отраженным от других объектов. В таком случае интенсивность может быть эмпирически выражена следующей формулой:

$$I_{отр} = I_a K_a + I (K_d \cos \theta + K_s \cos^p \alpha),$$

где I_a — интенсивность рассеянного света, K_a — константа.

Можно еще усовершенствовать модель отражения, если учесть то, что энергия от точечного источника света уменьшается пропорционально квадрату

расстояния. Использование такого правила вызывает сложности, поэтому н практике часто реализуют модель, выражаемую эмпирической формулой [28]:

$$I_{отр} = I_a K_a + \frac{I}{R+k}\left(K_d \cos\theta + K_s \cos^p \alpha\right),$$

где R — расстояние от центра проекции до поверхности, k — константа.

Как определить цвет закрашивания точек объектов в соответствии с данно моделью? Наиболее просто выполняется расчет в градациях серого цвет (например, для белого источника света и серых объектов). В данном случа интенсивность отраженного света соответствует яркости. Сложнее обстои дело с цветными источниками света, освещающими цветные поверхности Например, для модели RGB составляются три формулы расчета интенсивно сти отраженного света для различных цветовых компонент. Коэффициенть K_a и K_d различны для разных компонент — они выражают собственный цве поверхности. Поскольку цвет отраженного зеркального луча равен цвету ис точника, то коэффициент K_s будет одинаковым для всех компонент цветово модели. Цвет источника света выражается значениями интенсивности I дл соответствующих цветовых компонент.

Алгебра векторов

Здесь уместно сделать небольшое отступление от темы. Рассмотрим элементы ***алгебры векторов***. *Вектором* называется отрезок прямой, соединяющий некоторые точки пространства A и B. Направление вектора — от начальной точки A к конечной точке B. *Радиус-вектор* $\boldsymbol{R}$ — это вектор, с начальной точкой в центре координат. Координатами радиус-вектора являются координаты конечной точки (рис. 4.31). Длина радиус-вектора часто называется *модулем*, обозначается как $|R|$ и вычисляется следующим образом:

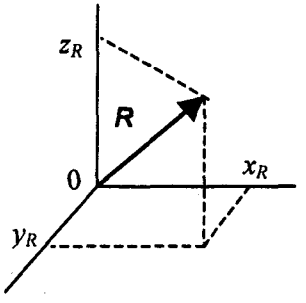


Рис. 4.31. Радиус-вектор

$$|R| = \sqrt{x_R^2 + y_R^2 + z_R^2}.$$

Единичный вектор — это вектор, длина которого равна единице. Перечислим основные операции над векторами.

1. *Умножение вектора на число* $X = Va$. Результат — вектор X, длина которого в a раз больше вектора V. Если число a положительно, то направление вектора X совпадает с вектором V. При $a<0$ вектор X имеет противо-

положное направление вектору V. Если V — это радиус-вектор, то координаты вектора результата будут ($a \cdot x_V$, $a \cdot y_V$, $a \cdot z_V$), то есть каждая координата вектора умножается на число a.

2. *Сложение векторов* $C = A+B$. Результат сложения — это вектор, соответствующий одной из диагоналей параллелограмма, сторонами которого являются векторы A и B (рис. 4.32). Все три вектора лежат в одной плоскости. Для радиус-векторов A и B координаты вектора результата определяются так:

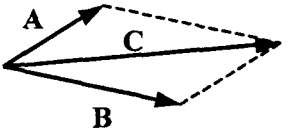


Рис. 4.32. Сложение векторов

$$x_C = x_A + x_B,$$
$$y_C = y_A + y_B,$$
$$z_C = z_A + z_B.$$

Разность двух векторов $C = A - B$ можно определить через операцию сложения $C = A + (-B)$. Вектор разности соответствует другой диагонали параллелограмма, изображенного на рис. 4.32. При вычитании радиус-векторов соответствующие координаты вычитаются:

$$x_C = x_A - x_B,$$
$$y_C = y_A - y_B,$$
$$z_C = z_A - z_B.$$

3. *Скалярное произведение векторов* $c = A \cdot B$. Результатом операции является число (скаляр), которое равно произведению длин векторов на косинус угла между ними:

$$c = A \cdot B = |A|\,|B| \cos \varphi.$$

Если A и B — это радиус-векторы, то результат можно вычислить через координаты следующим образом:

$$c = x_A \cdot x_B + y_A \cdot y_B + z_A \cdot z_B.$$

4. *Векторное произведение векторов* $C = A \times B$. Результатом операции является вектор, перпендикулярный к плоскости параллелограмма, образованного сторонами векторов A и B, а длина вектора равна площади этого параллелограмма — подобно тому, как изображено на рис. 4.33.

$$|C| = |A|\,|B| \sin \varphi.$$

В случае, когда векторы A и B являются радиус-векторами, координаты вектора результата C вычисляются по формулам:

$$x_C = y_A z_B - z_A y_B,$$
$$y_C = z_A x_B - x_A z_B,$$
$$z_C = x_A y_B - y_A x_B.$$

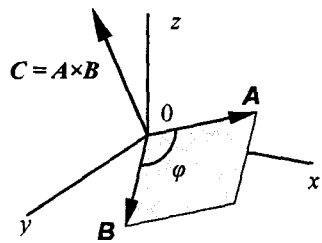


Рис. 4.33. Векторное произведение

Обратите внимание на то, что $A \times B = -B \times A$. Иными словами, порядок сомножителей определяет направление вектора результата. В этом можно убедиться, если в формуле координат поменять местами координаты векторов A и B.

Кроме того, направление вектора результата операции $A \times B$ зависит и от расположения координатных осей (система координат, изображенная на рис. 4.33, называется *левой* [16]). Назовите ось y осью x, а ось x — осью y (получится *правая* система), а также соответственно поменяйте местами координаты x и y векторов A и B в формуле векторного произведения. В результате такой перестановки координаты вектора C поменяют знак, то есть вектор будет иметь противоположное направление.

Вычисление нормалей и углов отражения

Вычисление координат вектора нормали. Рассматривая модели отражения света, вы, наверное, обратили внимание на то, что нормаль к поверхности является важным элементом. Определение вектора нормали к поверхности в заданной точке может быть выполнено различными способами. В значительной степени это определяется типом модели описания поверхности. Для поверхностей, заданных в аналитической форме, известны методы дифференциальной геометрии, которые основываются на вычислении частных производных функций описания [19]. Например, если поверхность задана параметрическими функциями

$$x = x(s, t),$$
$$y = y(s, t),$$
$$z = z(s, t),$$

то координаты вектора нормали можно вычислить так [3]:

$$x_N = \begin{Vmatrix} \frac{\partial y}{\partial s} & \frac{\partial z}{\partial s} \\ \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{Vmatrix} = \frac{\partial y}{\partial s} \cdot \frac{\partial z}{\partial t} - \frac{\partial y}{\partial t} \cdot \frac{\partial z}{\partial s},$$

$$y_N = \begin{Vmatrix} \frac{\partial z}{\partial s} & \frac{\partial x}{\partial s} \\ \frac{\partial z}{\partial t} & \frac{\partial x}{\partial t} \end{Vmatrix} = \frac{\partial z}{\partial s} \cdot \frac{\partial x}{\partial t} - \frac{\partial z}{\partial t} \cdot \frac{\partial x}{\partial s},$$

$$z_N = \begin{Vmatrix} \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} \end{Vmatrix} = \frac{\partial x}{\partial s} \cdot \frac{\partial y}{\partial t} - \frac{\partial x}{\partial t} \cdot \frac{\partial y}{\partial s}.$$

В случае описания поверхности векторно-полигональной моделью для определения нормалей можно использовать методы векторной алгебры.

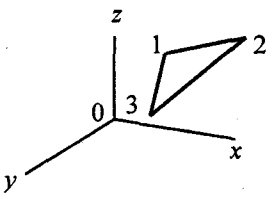


Рис. 4.34. Одна грань поверхности

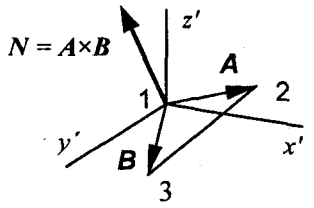


Рис. 4.35. Радиус-векторы

Пусть в пространстве задана некоторая многогранная поверхность. Рассмотрим одну ее плоскую грань в виде треугольника (рис. 4.34). Для вычисления координат вектора нормали воспользуемся векторным произведением любых двух векторов, лежащих в плоскости грани. В качестве таких векторов могут служить и ребра грани, например, ребра 1-2 и 1-3. Однако формулы для векторного произведения были определены нами только для радиус-векторов. Чтобы перейти к радиус-векторам, введем новую систему координат, центр которой совпадает с вершиной 1, а оси параллельны осям прежней системы. Координаты вершин в новой системе:

$$x'_i = x_i - x_1,$$
$$y'_i = y_i - y_1,$$
$$z'_i = z_i - z_1.$$

Теперь назовем ребро (1-2) вектором A, а ребро (1-3) — вектором B, как показано на рис. 4.35. Таким образом, положение нормали к грани в пространстве будет описываться радиус-вектором N. Его координаты в системе (x', y', z') выразим формулами для векторного произведения:

$$x'_N = (y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1);$$
$$y'_N = (z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1);$$
$$z'_N = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

Здесь использованы координаты вершин грани до переноса.

Плоская грань может изображаться в различных ракурсах. В каждой кон кретной ситуации необходимо выбирать направление нормали, соответст вующее *видимой стороне* грани. Если плоская грань может быть видна с об ратной стороны, то тогда в расчетах отраженного света необходимо выбират в качестве нормали обратный вектор, то есть (–*N*).

Если полигональная поверхность имеет не треугольные грани, а, например плоские четырехугольные, то расчет нормали можно выполнять по любы трем вершинам грани.

Диффузное отражение. Определим косинус угла между вектором нормали направлением на источник света.

Первый пример (возможно, самый простой). Источник света располагается н оси z в бесконечности. Если расчеты производятся для видовой системы ко ординат, то это означает, что источник света располагается на одной ос с камерой.

Косинус угла нормали к грани с осью z равен отношению координаты и длины радиус-вектора

$$\cos\theta = \frac{z_N}{|N|} = \frac{z_N}{\sqrt{x_N^2 + y_N^2 + z_N^2}}.$$

Второй пример. Источник света располагается в бесконечности и не лежи на оси z. В этом случае существенным является способ задания направлени на источник света. Если расположение источника света описывать так же как и для камеры — двумя углами (α_c и β_c), то можно сделать поворот коор динат так, чтобы ось z была направлена на источник света, и применить фор мулы для первого примера. Иными словами, необходимо преобразовать ко ординаты вектора нормали. Здесь можно использовать тот факт, что длина вектора при повороте не изменяется, поэтому достаточно вычислить координ нату z_N в повернутой системе координат.

Если расположение источника света описывается вектором, направленным на источник света, то косинус угла с вектором нормали можно вычислить сле дующим образом. Вначале необходимо определить радиус-вектор, направ ленный на источник света. Обозначим его как S. Затем, для вычисления ко

синуса угла между радиус-векторами $\boldsymbol{S}$ и $\boldsymbol{N}$ воспользуемся формулами скалярного произведения векторов. Так как

$$\boldsymbol{S} \cdot \boldsymbol{N} = |S|\,|N| \cos\theta,$$

а также

$$\boldsymbol{S} \cdot \boldsymbol{N} = x_S\, x_N + y_S\, y_N + z_S\, z_N,$$

то получим

$$\cos\theta = \frac{x_S x_N + y_S y_N + z_S z_N}{|S||N|}.$$

Очевидно, что для упрощения вычислений целесообразно использовать вектор $\boldsymbol{S}$ единичной длины, то есть $|S| = 1$.

Третий пример. Источник света располагается в конечной точке пространства с координатами (x_c, y_c, z_c). Для определения косинуса угла с нормалью выполним сдвиг координат источника света так, чтобы вектор нормали в точке поверхности и вектор, направленный на источник света, выходили из общего центра. Выше мы уже рассматривали построение радиус-вектора нормали к треугольной грани путем сдвига (параллельного переноса) координат на $(-x_1, -y_1, -z_1)$. Радиус-вектор, который направлен на источник света и который можно использовать для расчетов, будет иметь координаты $(x_c - x_1,\ y_c - y_1,\ z_c - z_1)$. Затем уже можно вычислить искомый косинус угла через скалярное произведение радиус-векторов, как в предыдущем примере.

Зеркальное отражение. Будем считать, что задан радиус-вектор $\boldsymbol{S}$, направленный на источник света, а также известен радиус-вектор нормали $\boldsymbol{N}$. Требуется найти косинус угла между отраженным лучом и направлением камеры.

Вначале необходимо вычислить радиус-вектор отраженного луча. Обозначим его как $\boldsymbol{R}$. Выполним некоторые геометрические построения, как показано на рис. 4.36.

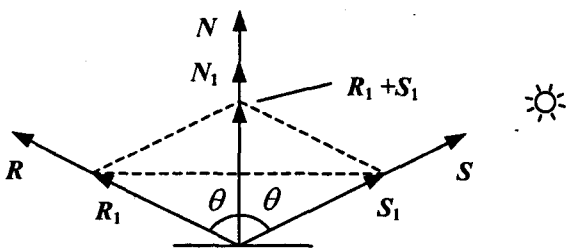


Рис. 4.36. Векторы $\boldsymbol{R}_1$, $\boldsymbol{S}_1$ и $\boldsymbol{N}_1$ — единичной длины

Для решения нашей задачи вначале рассмотрим единичные векторы $\boldsymbol{R}_1$, и $\boldsymbol{N}_1$. Поскольку векторы нормали, падающего луча и отраженного луча н ходятся в одной плоскости, то можно записать $\boldsymbol{R}_1 + \boldsymbol{S}_1 = \boldsymbol{N}'$, где $\boldsymbol{N}'$ — это ве тор, соответствующий диагонали ромба и совпадающий по направлению нормалью. Длина вектора $\boldsymbol{N}'$ равна $2\cos\theta$. Так как вектор $\boldsymbol{N}'$ по направлени совпадает с $\boldsymbol{N}_1$, то

$$\boldsymbol{N}' = \boldsymbol{N}_1 2\cos\theta,$$

или

$$\boldsymbol{R}_1 + \boldsymbol{S}_1 = \boldsymbol{N}_1 2\cos\theta.$$

Отсюда найдем единичный вектор отраженного луча:

$$\boldsymbol{R}_1 = \boldsymbol{N}_1 2\cos\theta - \boldsymbol{S}_1 = \frac{\boldsymbol{N}}{|N|} 2\cos\theta - \frac{\boldsymbol{S}}{|S|}.$$

Найдем $\cos\theta$. Это можно сделать, используя скалярное произведение векто ров $\boldsymbol{N}$ и $\boldsymbol{S}$:

$$\cos\theta = \frac{\boldsymbol{N}\cdot\boldsymbol{S}}{|N||S|}.$$

Подставим это значение в выражение для $\boldsymbol{R}_1$:

$$\boldsymbol{R}_1 = \boldsymbol{N}\cdot 2\frac{\boldsymbol{N}\cdot\boldsymbol{S}}{|N|^2|S|} - \frac{\boldsymbol{S}}{|S|}.$$

Полагая, что искомый вектор отраженного луча будет иметь такую же длин что и вектор падающего луча, то есть $\boldsymbol{R} = |S|\boldsymbol{R}_1$, получим

$$\boldsymbol{R} = \boldsymbol{N}\cdot 2\frac{\boldsymbol{N}\cdot\boldsymbol{S}}{|N|^2} - \boldsymbol{S}.$$

Это решение в векторной форме. Запишем координаты вектора $\boldsymbol{R}$.

$$x_R = 2x_N \frac{x_N x_S + y_N y_S + z_N z_S}{x_N^2 + y_N^2 + z_N^2} - x_S,$$

$$y_R = 2y_N \frac{x_N x_S + y_N y_S + z_N z_S}{x_N^2 + y_N^2 + z_N^2} - y_S,$$

$$z_R = 2z_N \frac{x_N x_S + y_N y_S + z_N z_S}{x_N^2 + y_N^2 + z_N^2} - z_S.$$

Теперь осталось найти косинус угла между отраженным лучом и направлением камеры. Обозначим радиус-вектор, направленный на камеру, как ***K***. Искомый косинус угла найдем, используя скалярное произведение векторов ***K*** и ***R***:

$$\cos\alpha = \frac{\boldsymbol{K}\cdot\boldsymbol{R}}{|K||R|} = \frac{x_K x_R + y_K y_R + z_K z_R}{\sqrt{x_K^2 + y_K^2 + z_K^2}\sqrt{x_R^2 + y_R^2 + z_R^2}}.$$

Очевидно, что для упрощения вычислений целесообразно задавать векторы ***S***, ***N*** и ***K*** единичной длины (тогда и ***R*** будет единичным).

Метод Гуро

Этот метод предназначен для создания иллюзии гладкой криволинейной поверхности, описанной в виде многогранников или полигональной сетки с плоскими гранями. Если каждая плоская грань имеет один постоянный цвет, определенный с учетом отражения, то различные цвета соседних граней очень заметны, и поверхность выглядит именно как многогранник. Казалось бы, этот дефект можно замаскировать за счет увеличения количества граней при аппроксимации поверхности. Но зрение человека имеет способность подчеркивать перепады яркости на границах смежных граней — такой эффект называется *эффектом полос Маха*. Поэтому для создания иллюзии гладкости нужно намного увеличить количество граней, что приводит к существенному замедлению визуализации — чем больше граней, тем меньше скорость рисования объектов.

Метод Гуро основывается на идее закрашивания каждой плоской грани не одним цветом, а плавно изменяющимися оттенками, вычисляемыми путем интерполяции цветов примыкающих граней. Закрашивание граней по методу Гуро осуществляется в четыре этапа.

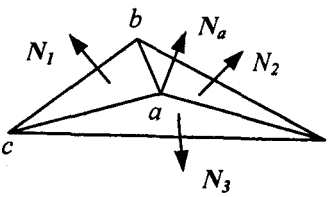


Рис. 4.37. Нормаль в вершине

- Вычисляются нормали к каждой грани.
- Определяются нормали в вершинах. Нормаль в вершине определяется усреднением нормалей примыкающих граней (рис. 4.37).
- На основе нормалей в вершинах вычисляются значения интенсивностей в вершинах согласно выбранной модели отражения света.
- Закрашиваются полигоны граней цветом, соответствующим линейной интерполяции значений интенсивности в вершинах.

Вектор нормали в вершине (a) равен

$$N_a = (N_1 + N_2 + N_3) / 3.$$

Определение интерполированных значений интенсивности отраженного света в каждой точке грани (и, следовательно, цвет каждого пиксела) удобно выполнять во время цикла заполнения полигона. Рассмотрим заполнение контура грани горизонталями в экранных координатах (рис. 4.38).

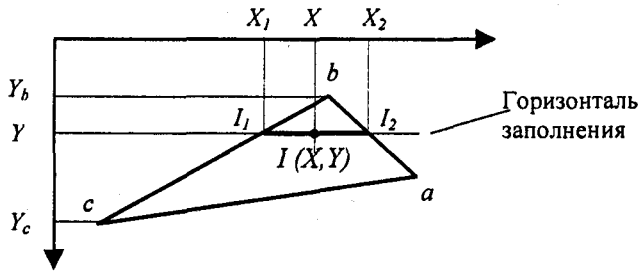


Рис. 4.38. Заполнение контура грани

Интерполированная интенсивность I в точке (X,Y) определяется исходя из пропорции $(I - I_1) / (X - X_1) = (I_2 - I_1) / (X_2 - X_1)$.

Отсюда $I = I_1 + (I_2 - I_1)(X - X_1) / (X_2 - X_1)$.

Значения интенсивностей I_1 и I_2 на концах горизонтального отрезка представляют собой интерполяцию интенсивности в вершинах:

$$(I_1 - I_b) / (Y - Y_b) = (I_c - I_b) / (Y_c - Y_b),$$
$$(I_2 - I_b) / (Y - Y_b) = (I_a - I_b) / (Y_a - Y_b)$$

или

$$I_1 = I_b + (I_c - I_b)(Y - Y_b) / (Y_c - Y_b),$$
$$I_2 = I_b + (I_a - I_b)(Y - Y_b) / (Y_a - Y_b).$$

Метод Фонга

Аналогичен методу Гуро, но при использовании метода Фонга для определения цвета в каждой точке интерполируются не интенсивности отраженного света, а векторы нормалей.

- Определяются нормали к граням.
- По нормалям к граням определяются нормали в вершинах. В каждой точке закрашиваемой грани определяется интерполированный вектор нормали.

- По направлению векторов нормали определяется цвет точек грани в соответствии с выбранной моделью отражения света.

Рассмотрим, как можно получить вектор нормали в каждой точке грани. Для интерполяции будем оперировать векторами N'_a, N'_b и N'_c, исходящими из центра координат плоскости проецирования и параллельными соответствующим нормалям N_a, N_b и N_c в вершинах a, b и c (рис. 4.39).

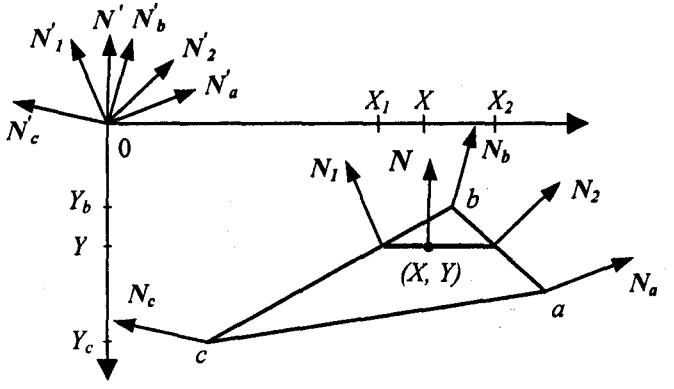


Рис. 4.39. Интерполяция векторов нормалей

Сначала найдем N'_1 и N'_2:

$$N'_1 = \begin{bmatrix} X_{N1} \\ Y_{N1} \\ Z_{N1} \end{bmatrix} = \begin{bmatrix} X_{Nb} + (X_{Nc} - X_{Nb})(Y - Y_b)/(Y_c - Y_b) \\ Y_{Nb} + (Y_{Nc} - Y_{Nb})(Y - Y_b)/(Y_c - Y_b) \\ Z_{Nb} + (Z_{Nc} - Z_{Nb})(Y - Y_b)/(Y_c - Y_b) \end{bmatrix},$$

$$N'_2 = \begin{bmatrix} X_{N2} \\ Y_{N2} \\ Z_{N2} \end{bmatrix} = \begin{bmatrix} X_{Nb} + (X_{Na} - X_{Nb})(Y - Y_b)/(Y_a - Y_b) \\ Y_{Nb} + (Y_{Na} - Y_{Nb})(Y - Y_b)/(Y_a - Y_b) \\ Z_{Nb} + (Z_{Na} - Z_{Nb})(Y - Y_b)/(Y_a - Y_b) \end{bmatrix}.$$

где X_{Na}, Y_{Na}, Z_{Na}, X_{Nb}, Y_{Nb}, Z_{Nb}, X_{Nc}, Y_{Nc} и Z_{Nc} — координаты векторов N'_a, N'_b и N'_c. Теперь найдем координаты вектора N':

$$N' = \begin{bmatrix} X_N \\ Y_N \\ Z_N \end{bmatrix} = \begin{bmatrix} X_{N1} + (X_{N2} - X_{N1})(X - X_1)/(X_2 - X_1) \\ Y_{N1} + (Y_{N2} - Y_{N1})(X - X_1)/(X_2 - X_1) \\ Z_{N1} + (Z_{N2} - Z_{N1})(X - X_1)/(X_2 - X_1) \end{bmatrix}.$$

Вектор N' параллелен вектору N для нормали в точке (X, Y), поэтому его можно использовать для расчета отражения света так же, как и вектор нормали N.

Метод Фонга сложнее, чем метод Гуро. Для каждой точки (пиксела) поверхности необходимо выполнять намного больше вычислительных операций. Тем не менее он дает значительно лучшие результаты, в особенности при имитации зеркальных поверхностей.

Общие черты и отличия методов Гуро и Фонга можно показать на примере цилиндрической поверхности, аппроксимированной многогранником (рис. 4.40). Пусть источник света находится позади нас. Проанализируем закрашивания боковых граней цилиндра.

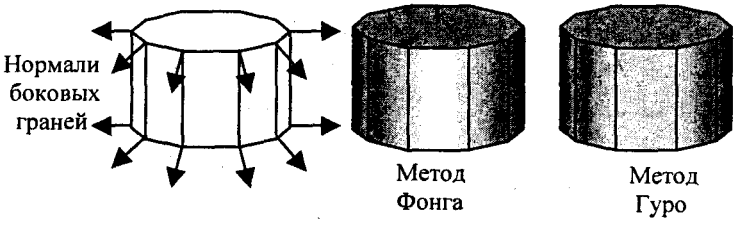


Рис. 4.40. Отличия закрашивания Фонга и Гуро

На рис. 4.40 на закрашенной поверхности показаны черным цветом ребра граней — это сделано для иллюстрации особенностей закрашивания, на самом деле после закрашивания никакого черного каркаса не будет, и поверхность выглядит гладкой.

Основные отличия можно заметить для закрашивания передней грани. Она перпендикулярна направлению лучей света. Поэтому нормали в вершинах этой грани располагаются симметрично — они образовывают попарно равные по абсолютной величине углы с лучами света. Для метода Гуро это обуславливает одинаковые интенсивности в вершинах передней грани. А раз интенсивности одинаковые, то и для любой точки внутри этой грани интенсивность одинакова (для линейной интерполяции). Это обуславливает единый цвет закрашивания. **Все точки передней грани имеют одинаковый цвет**, что, очевидно, неправильно.

Метод Фонга дает правильное закрашивание. Если интерполировать векторы нормалей передней грани, то по центру будут интерполированные нормали, параллельные лучам света (рис. 4.41).

По методу Фонга центр передней грани будет светлее, чем края. Возможно, это не очень заметно на типографском отпечатке рисунка данной книги, однако это именно так.

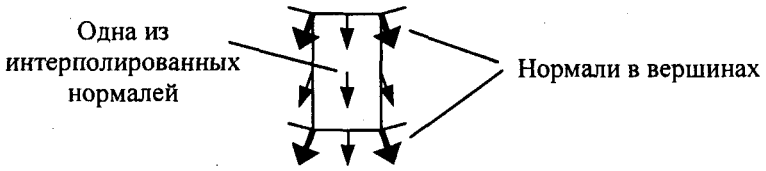


Рис. 4.41. Интерполяция нормалей дает более точный результат

Преломление света

Законы преломления света следует учитывать при построении изображений прозрачных объектов.

Модель идеального преломления. Согласно этой модели луч отклоняется на границе двух сред, причем падающий луч, преломленный луч и нормаль лежат в одной плоскости (в этой же плоскости лежит и зеркально отраженный луч). Обозначим угол между падающим лучом и нормалью как α_1, а угол между нормалью и преломленным лучом как α_2. Для этих углов известен закон *Снеллиуса*, согласно которому

$$n_1 \sin \alpha_1 = n_2 \sin \alpha_2,$$

где n_1 и n_2 — *абсолютные показатели преломления* соответствующих сред. На рис. 4.42 изображен пример отклонения луча при преломлении. В данном случае границами раздела сред являются две параллельные плоскости, например, при прохождении луча через толстое стекло. Очевидно, что угол α_1 равен углу α_4, а угол α_2 равен углу α_3. Иными словами, после прохождения сквозь стекло луч параллельно смещается. Это смещение зависит от толщины стекла и соотношения показателей преломления сред. Возможно, это самый простой пример преломления. Вы наверняка уже наблюдали и более сложные объекты, например треугольную призму. Для нее границами сред

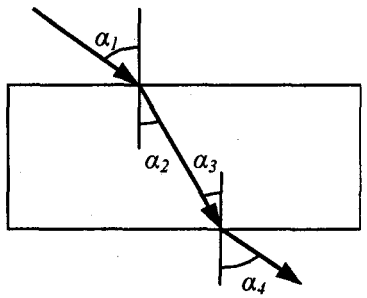


Рис. 4.42. Преломление луча

являются непараллельные плоскости (рис. 4.43). Прозрачные объекты могут иметь и криволинейные поверхности, например линзы в разнообразных оптических приборах.

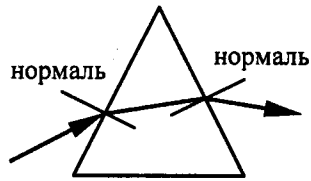


Рис. 4.43. Преломление в треугольной призме

Принято считать, что для вакуума абсолютный показатель преломления равен единице. Для воздуха он составляет 1.00029, для воды — 1.33, для стекла разных сортов: 1.52 (легкий крон), 1.65 (тяжелый крон) [13]. Показатель преломления зависит от состояния вещества, например, от температуры. На практике обычно используют отношение показателей преломления двух сред (n_1 / n_2), называемое *относительным показателем преломления.*

Еще одним важным аспектом преломления является зависимость отклонения луча от длины волны. Это наблюдалось еще *И. Ньютоном* в опытах по разложению белого света треугольной призмой (рис. 4.44).

Чем меньше длина волны, тем больше отклоняется луч при преломлении. Благодаря этому свойству преломления мы и наблюдаем радугу. Фиолетовый (λ=0.4 мкм) луч отклоняется больше всего, а красный (λ=0.7 мкм) — меньше всего. Например, для стекла показатель преломления в видимом спектре изменяется от 1.53 до 1.51 [13].

Таким образом, каждый прозрачный материал описывается показателем преломления, зависящим от длины волны. Кроме того, необходимо учитывать, какая часть световой энергии отражается, а какая часть проходит через объект и описывается преломлением света.

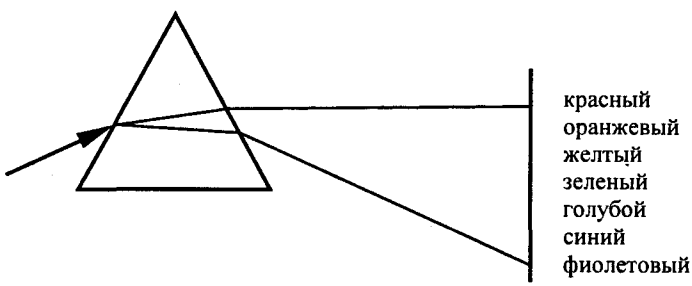


Рис. 4.44. Преломление зависит от длины волны

Кроме идеального преломления в компьютерной графике (хотя и значительно реже, вследствие сложности реализации) используется ***диффузное преломление*** [32]. Согласно этой модели падающий луч преломляется во все стороны. Примером может служить молочное стекло, обледеневшее стекло.

Вычисление вектора преломленного луча

Сформулируем задачу следующим образом. Заданы два единичных вектора: S_1 — это радиус-вектор, направленный на источник, и N_1 — радиус-вектор нормали к границе раздела двух сред. Также должны быть известны два коэффициента преломления для данных сред — n_1 и n_2 (или же их отношение).

Требуется найти единичный радиус-вектор преломленного луча T_1. Для решения этой задачи выполним некоторые геометрические построения (рис. 4.45). Нанесем еще несколько радиус-векторов (далее мы их все для краткости будем называть векторами).

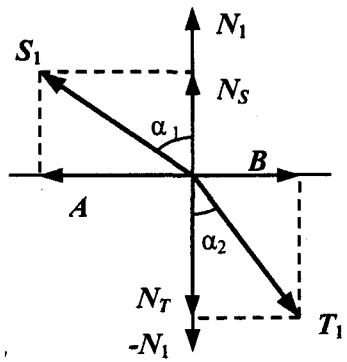


Рис. 4.45. Преломление

Искомый вектор T_1 равен сумме двух векторов: $T_1 = N_T + B$.

Найдем вначале вектор N_T. Он противоположен по направлению вектору нормали, а его длина равна $|T_1| \cos \alpha_2 = \cos \alpha_2$ (поскольку по условию задачи T_1 — единичный). Таким образом, $N_T = -N_1 \cos \alpha_2$. Необходимо определить $\cos \alpha_2$. Запишем закон преломления $n_1 \sin \alpha_1 = n_2 \sin \alpha_2$ в виде:

$$\sin \alpha_2 = n \sin \alpha_1,$$

где $n = n_1 / n_2$.

Воспользуемся тождеством $\cos^2 + \sin^2 = 1$. Тогда

$$\cos\alpha_2 = \sqrt{1 - \sin^2 \alpha_2} = \sqrt{1 - n^2 \sin^2 \alpha_1}$$

или

$$\cos\alpha_2 = \sqrt{1 + n^2 \left(\cos^2 \alpha_1 - 1\right)}.$$

Значение $\cos \alpha_1$ можно выразить через скалярное произведение единичных векторов S_1 и N_1, то есть $\cos \alpha_1 = S_1 \cdot N_1$. Тогда мы можем записать такое выражение для вектора N_T:

$$N_T = -N_1 \sqrt{1 + n^2 \left((S_1 \cdot N_1)^2 - 1\right)}.$$

Осталось найти выражение для вектора $\boldsymbol{B}$. Он располагается на одной прямой с вектором $\boldsymbol{A}$, причем $\boldsymbol{A} = \boldsymbol{S}_1 - \boldsymbol{N}_S$. Учитывая, что вектор $\boldsymbol{N}_S$ равен $\boldsymbol{N}_1 \cos\alpha_1$, то $\boldsymbol{A}=\boldsymbol{S}_1 - \boldsymbol{N}_1 \cos\alpha_1$. Так как $\cos\alpha_1=\boldsymbol{S}_1 \cdot \boldsymbol{N}_1$, то $\boldsymbol{A} = \boldsymbol{S}_1 - \boldsymbol{N}_1 (\boldsymbol{S}_1 \cdot \boldsymbol{N}_1)$.

Поскольку длина вектора $\boldsymbol{A}$ равна $\sin\alpha_1$, а длина вектора $\boldsymbol{B}$ равна $\sin\alpha_2$, то

$$\frac{|\boldsymbol{B}|}{|\boldsymbol{A}|} = \frac{\sin\alpha_2}{\sin\alpha_1} = \frac{n_2}{n_1} = n,$$

откуда $|B| = n\,|A|$. Учитывая взаимное расположение векторов $\boldsymbol{A}$ и $\boldsymbol{B}$, получим $\boldsymbol{B} = -n\boldsymbol{A} = n(\boldsymbol{N}_1 (\boldsymbol{S}_1 \cdot \boldsymbol{N}_1) - \boldsymbol{S}_1)$.

Теперь мы можем записать искомое выражение для единичного радиус-вектора луча преломления $\boldsymbol{T}_1$:

$$\boldsymbol{T}_1 = n\boldsymbol{N}_1(\boldsymbol{S}_1 \cdot \boldsymbol{N}_1) - n\boldsymbol{S}_1 - \boldsymbol{N}_1\sqrt{1 + n^2\left((\boldsymbol{S}_1 \cdot \boldsymbol{N}_1)^2 - 1\right)}.$$

Если подкоренное выражение отрицательно, то преломленный луч не существует. Это соответствует так называемому *полному внутреннему отражению* [13, 30, 32].

Это решение задачи в векторном виде. Используя полученное выражение, можно записать координаты вектора $\boldsymbol{T}_1$. Сделайте это самостоятельно в качестве упражнения.

Трассировка лучей

Методы трассировки лучей на сегодняшний день считаются наиболее мощными и универсальными методами создания реалистичных изображений. Известно много примеров реализации алгоритмов трассировки для качественного отображения самых сложных трехмерных сцен. Можно отметить, что универсальность методов трассировки в значительной степени обусловлена тем, что в их основе лежат простые и ясные понятия, отражающие наш опыт восприятия окружающего мира.

Как мы видим окружающую нас реальность? Во-первых, нужно определиться с тем, что мы вообще способны видеть. Это изучается в специальных дисциплинах, а в некоторой степени, это вопрос философский. Но здесь мы будем полагать, что окружающие нас объекты обладают по отношению к свету такими свойствами:

- излучают;
- отражают и поглощают;
- пропускают сквозь себя.

Каждое из этих свойств можно описать некоторым набором характеристик. Например, излучение можно охарактеризовать интенсивностью, направленностью, спектром. Излучение может исходить от условно точечного источника (далекая звезда) или протяженного (скажем, от извергающейся из кратера вулкана расплавленной лавы). Распространение излучения может осуществляться вдоль достаточно узкого луча (сфокусированный луч лазера), конусом (прожектор), равномерно во все стороны (Солнце), либо еще как-нибудь. Свойство отражения (поглощения) можно описать характеристиками диффузного рассеивания и зеркального отражения. Прозрачность можно описать ослаблением интенсивности и преломлением.

Распределение световой энергии по возможным направлениям световых лучей можно отобразить с помощью векторных диаграмм, в которых длина векторов соответствует интенсивности (рис. 4.46—4.48).

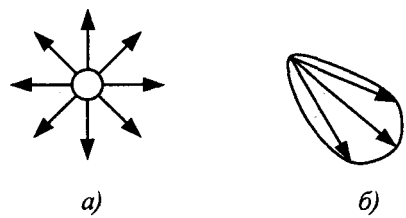


Рис. 4.46. Излучение: *а* — равномерно во все стороны, *б* — направленно

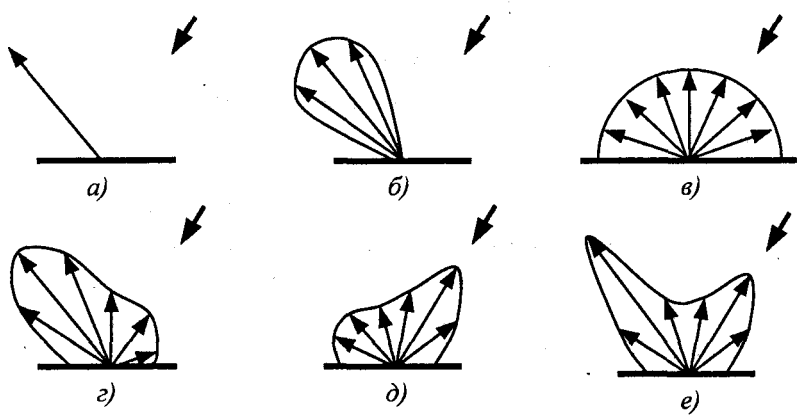


Рис. 4.47. Отражение: *а* — идеальное зеркало, *б* — неидеальное зеркало, *в* — диффузное, *г* — сумма диффузного и зеркального, *д* — обратное, *е* — сумма диффузного, зеркального и обратного

В предыдущих разделах мы с вами уже ознакомились с наиболее часто упоминаемыми видами отражения — зеркальным и диффузным. Реже в литературе упоминается обратное, антизеркальное отражение, у которого максимум интенсивности отражения соответствует направлению на источник. Обратным зеркальным отражением обладают некоторые виды растительности на поверхности Земли, наблюдаемые с высоты, например, рисовые поля [9].

Два крайних, идеализированных случая преломления изображены на рис. 4.48.

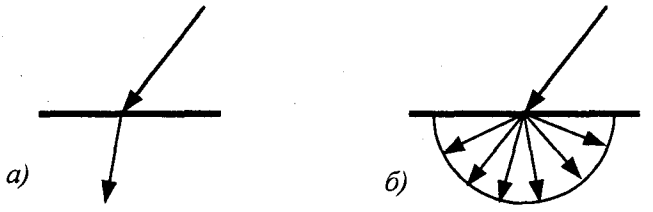


Рис. 4.48. Преломление: *а* — идеальное, *б* — диффузное

Некоторые реальные объекты преломляют лучи гораздо более сложным образом, например, обледеневшее стекло.

Один и тот же объект реальной действительности может восприниматься в виде источника света, а при ином рассмотрении может считаться предметом, только отражающим и пропускающим свет. Например, купол облачного неба в некоторой трехмерной сцене может моделироваться в виде протяженного (распределенного) источника света, а в других моделях это же небо выступает как полупрозрачная среда, освещенная со стороны Солнца.

В общем случае каждый объект описывается некоторым сочетанием перечисленных выше трех свойств. В качестве упражнения попробуйте привести пример объекта, который обладает одновременно тремя указанными свойствами — сам излучает свет и, в то же время, отражает, а также пропускает свет от других источников. Вероятно, ваше воображение подскажет иные примеры, нежели, скажем, докрасна раскаленное стекло.

Теперь рассмотрим то, как формируется изображение некоторой сцены, включающей в себя несколько пространственных объектов. Будем полагать, что из точек поверхности (объема) излучающих объектов исходят лучи света. Можно назвать такие лучи первичными — они освещают все остальное. Важным моментом является предположение, что световой луч в свободном пространстве распространяется ***вдоль прямой линии*** (хотя в специальных разделах физики изучаются также и причины возможного искривления). Но

в *геометрической оптике* полагают, что луч света распространяется прямолинейно до тех пор, пока не встретится отражающая поверхность или граница среды преломления. Так будем полагать и мы.

От источников излучения исходит по различным направлениям бесчисленное множество первичных лучей (даже луч лазера невозможно идеально сфокусировать — все равно свет будет распространяться не одной идеально тонкой линией, а конусом, пучком лучей). Некоторые лучи уходят в свободное пространство, а некоторые (их также бесчисленное множество) попадают на другие объекты. Если луч попадает в прозрачный объект, то, преломляясь, он идет дальше, при этом некоторая часть световой энергии поглощается. Подобно этому, если на пути луча встречается зеркально отражающая поверхность, то он также изменяет направление, а часть световой энергии поглощается. Если объект зеркальный и одновременно прозрачный (например, обычное стекло), то будет уже два луча — в этом случае говорят, что луч расщепляется.

Можно сказать, что в результате действия на объекты первичных лучей возникают вторичные лучи. Бесчисленное множество вторичных лучей уходит в свободное пространство, но некоторые из них попадают на другие объекты. Так, многократно отражаясь и преломляясь, отдельные световые лучи приходят в точку наблюдения — глаз человека или оптическую систему камеры. Очевидно, что в точку наблюдения может попасть и часть первичных лучей непосредственно от источников излучения. Таким образом, изображение сцены формируется некоторым множеством световых лучей.

Цвет отдельных точек изображения определяется спектром и интенсивностью первичных лучей источников излучения, а также поглощением световой энергии в объектах, встретившихся на пути соответствующих лучей.

Непосредственная реализация данной лучевой модели формирования изображения представляется затруднительной. Можно попробовать построить алгоритм построения изображения указанным способом. В таком алгоритме необходимо предусмотреть перебор всех первичных лучей и определить те из них, которые попадают в объекты и в камеру. Затем выполнить перебор всех вторичных лучей, и также учесть только те, которые попадают в объекты и в камеру. И так далее. Можно назвать такой метод *прямой* трассировкой лучей. Практическая ценность такого метода вызывает сомнения. В самом деле, как учитывать бесконечное множество лучей, идущих во все стороны? Очевидно, что полный перебор бесконечного числа лучей в принципе невозможен. Даже если каким-то образом свести это к конечному числу операций (например, разделить всю сферу направлений на угловые секторы и оперировать уже не бесконечно тонкими линиями, а секторами), все равно остается главный не-

достаток метода — много лишних операций, связанных с расчетом лучей, которые затем не используются. Так, во всяком случае, это представляется в настоящее время.

Метод ***обратной трассировки*** лучей позволяет значительно сократить перебор световых лучей. Метод разработан в 80-х годах, основополагающими считаются работы *Уиттеда* и *Кэя* [28]. Согласно этому методу отслеживание лучей производится не от источников света, а в обратном направлении — от точки наблюдения. Так учитываются только те лучи, которые вносят вклад в формирование изображения.

Рассмотрим, как можно получить растровое изображение некоторой трехмерной сцены методом обратной трассировки. Предположим, что плоскость проецирования разбита на множество квадратиков — пикселов. Выберем центральную проекцию с центром схода на некотором расстоянии от плоскости проецирования. Проведем прямую линию из центра схода через середину квадратика (пиксела) плоскости проецирования (рис. 4.49). Это будет первичный луч обратной трассировки. Если прямая линия этого луча попадает в один или несколько объектов сцены, то выбираем ближайшую точку пересечения. Для определения цвета пиксела изображения нужно учитывать свойства объекта, а также то, какое световое излучение приходится на соответствующую точку объекта.

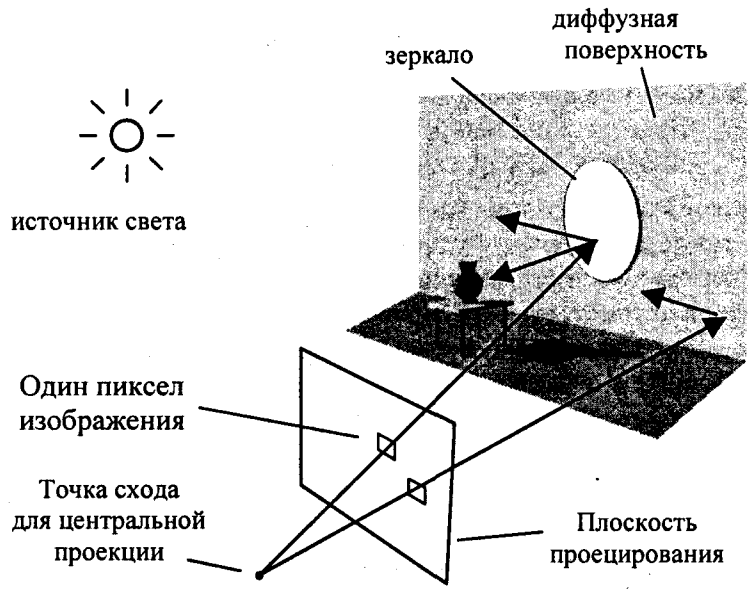


Рис. 4.49. Схема обратной трассировки лучей

Если объект зеркальный (хотя бы частично), то строим вторичный луч — луч падения, считая лучом отражения предыдущий, первичный трассируемый луч. Выше мы рассматривали зеркальное отражение и получили формулы для вектора отраженного луча по заданным векторам нормали и луча падения. Но здесь нам известен вектор отраженного луча, а как найти вектор падающего луча? Для этого можно использовать ту же самую формулу зеркального отражения, но определяя необходимый вектор луча падения как отраженный луч. То есть отражение наоборот.

Для идеального зеркала достаточно затем проследить лишь очередную точку пересечения вторичного луча с некоторым объектом. Что означает термин "идеальное зеркало"? Будем полагать, что у такого зеркала идеально ровная отполированная поверхность, поэтому одному отраженному лучу соответствует только один падающий луч. Зеркало может быть затемненным, то есть поглощать часть световой энергии, но все равно остается правило: один луч падает — один отражается. Можно рассматривать также "неидеальное зеркало". Это будет означать, что поверхность неровная. Направлению отраженного луча будет соответствовать несколько падающих лучей (или наоборот, один падающий луч порождает несколько отраженных лучей), образующих некоторый конус, возможно, несимметричный, с осью вдоль линии падающего луча идеального зеркала. Конус соответствует некоторому закону распределения интенсивностей, простейший из которых описывается моделью Фонга — косинус угла, возведенный в некоторую степень. Неидеальное зеркало резко усложняет трассировку — нужно проследить не один, а множество падающих лучей, учитывать вклад излучения от других видимых из данной точки объектов.

Если объект прозрачный, то необходимо построить новый луч, такой, который при преломлении давал бы предыдущий трассируемый луч. Здесь также можно воспользоваться обратимостью, которая справедлива и для преломления. Для расчета вектора искомого луча можно применить рассмотренные выше формулы для вектора луча преломления, считая, что преломление происходит в обратном направлении (рис. 4.50).

Если объект обладает свойствами диффузного отражения и преломления, то, в общем случае, как и для неидеального зеркала, необходимо трассировать лучи, приходящие от всех имеющихся объектов. Для диффузного отражения интенсивность отраженного света, как известно, пропорциональна косинусу угла между вектором луча от источника света и нормалью. Здесь источником света может выступать любой видимый из данной точки объект, способный передавать световую энергию.

Когда выясняется, что текущий луч обратной трассировки не пересекает какой-либо объект, а уходит в свободное пространство, то на этом трассировка для этого луча заканчивается.

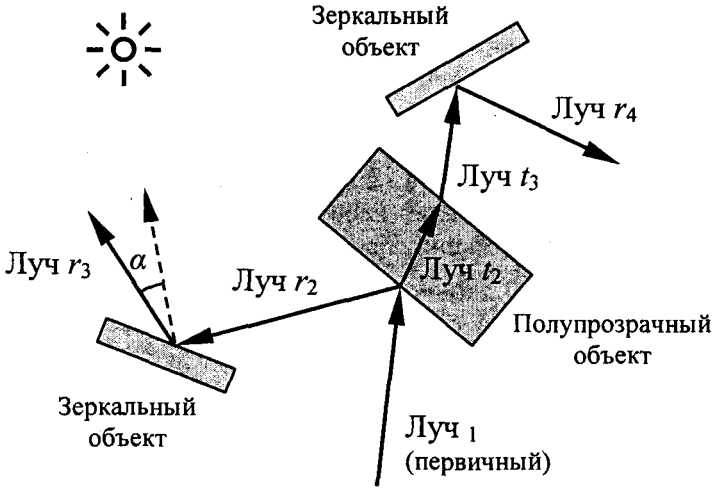


Рис. 4.50. Обратная трассировка для объектов, обладающих свойствами зеркального отражения и преломления

Обратная трассировка лучей в том виде, в каком мы ее здесь рассмотрели, хоть и сокращает перебор, но не позволяет избавиться от бесконечного числа анализируемых лучей. В самом деле, данный метод позволяет сразу получить для каждой точки изображения единственный первичный луч обратной трассировки. Однако вторичных лучей отражения уже может быть бесконечное количество. Так, например, если объект может отражать свет от любого другого объекта, и если эти другие объекты имеют достаточно большие размеры, то какие именно точки излучающих объектов нужно учитывать для построения соответствующих лучей, например, при диффузном отражении? Очевидно, все точки.

При практической реализации метода обратной трассировки вводят ограничения. Некоторые их них необходимы, чтобы можно было в принципе решить задачу синтеза изображения, а некоторые ограничения позволяют значительно повысить быстродействие трассировки. Рассмотрим примеры таких ограничений [28, 32].

1. Среди всех типов объектов выделим некоторые, которые назовем *источниками света*. Источники света могут только излучать свет, но не могут его отражать или преломлять. Будем рассматривать только *точечные* источники света.
2. Свойства отражающих поверхностей описываются суммой двух компонент — диффузной и зеркальной.

3. В свою очередь, зеркальность также описывается двумя составляющими. Первая (*reflection*) учитывает отражение от других объектов, не являющихся источниками света. Строится только один зеркально отраженный луч ***r*** для дальнейшей трассировки. Вторая компонента (*specular*) означает световые блики от источников света. Для этого направляются лучи на все источники света и определяются углы, образуемые этими лучами с зеркально отраженным лучом обратной трассировки (***r***). При зеркальном отражении цвет точки поверхности определяется цветом того, что отражается. В простейшем случае зеркало не имеет собственного цвета поверхности.
4. При диффузном отражении учитываются только лучи от источников света. Лучи от зеркально отражающих поверхностей игнорируются. Если луч, направленный на данный источник света, закрывается другим объектом, значит, данная точка объекта находится в тени. При диффузном отражении цвет освещенной точки поверхности определяется собственным цветом поверхности и цветом источников света.
5. Для прозрачных (*transparent*) объектов обычно не учитывается зависимость коэффициента преломления от длины волны. Иногда прозрачность вообще моделируют без преломления, то есть направление преломленного луча ***t*** совпадает с направлением падающего луча.
6. Для учета освещенности объектов светом, рассеиваемым другими объектами, вводится фоновая составляющая (*ambient*).
7. Для завершения трассировки вводят некоторое пороговое значение освещенности, которое уже не должно вносить вклад в результирующий цвет, либо ограничивают количество итераций.

Согласно модели Уиттеда цвет некоторой точки объекта определяется суммарной интенсивностью

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d I_d(\lambda) C(\lambda) + K_s I_s(\lambda) + K_r I_r(\lambda) + K_t I_t(\lambda),$$

где λ — длина волны, $C(\lambda)$ — заданный исходный цвет точки объекта, K_a, K_d, K_s, K_r и K_t — коэффициенты, учитывающие свойства конкретного объекта параметрами фоновой подсветки, диффузного рассеивания, зеркальности, отражения и прозрачности.

I_a — интенсивность фоновой подсветки,

I_d — интенсивность, учитываемая для диффузного рассеивания,

I_s — интенсивность, учитываемая для зеркальности,

I_r — интенсивность излучения, приходящего по отраженному лучу,

I_t — интенсивность излучения, приходящего по преломленному лучу.

Интенсивность фоновой подсветки (I_a) для некоторого объекта обычно константа. Запишем формулы для остальных интенсивностей. Для диффузного отражения:

$$I_d = \sum_i I_i(\lambda)\cos\theta_i,$$

где $I_i(\lambda)$ — интенсивность излучения i-го источника света, θ_i — угол между нормалью к поверхности объекта и направлением на i-й источник света.

Для зеркальности:

$$I_s = \sum_i I_i(\lambda)\cos^p\alpha_i,$$

где p — показатель степени от единицы до нескольких сотен (согласно модели Фонга), α_i — угол между отраженным лучам (обратной трассировки) и направлением на i-й источник света.

Интенсивности излучений, приходящих по отраженному лучу (I_r), а также по преломленному лучу (I_t), умножают на коэффициент, учитывающий ослабление интенсивности в зависимости от расстояния, пройденного лучом. Такой коэффициент записывается в виде $e^{-\beta d}$, где d — пройденное расстояние, β — параметр ослабления, учитывающий свойства среды, в которой распространяется луч.

На рис. 4.51 показан пример изображения трехмерной сцены.

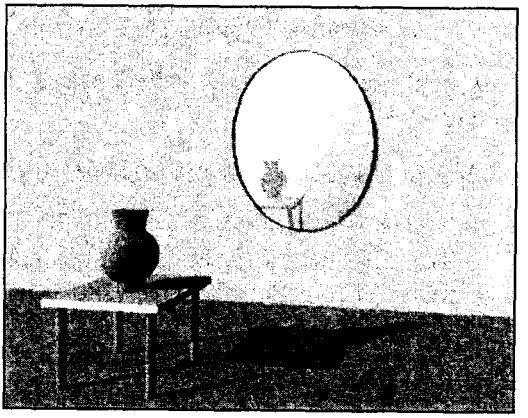


Рис. 4.51. Это изображение получено обратной трассировкой лучей

Запишем базовую операцию обратной трассировки лучей в виде функции, которую назовем ЛУЧ. Она вычисляет значение интенсивности для текущего

трассируемого луча. Для описания функции используем псевдокод, близкий по синтаксису языку С.

```
интенсивность ЛУЧ(номер итерации ind,   тип луча,
                  направление луча dir,  номер объекта no)
{
Находим точку пересечения луча с ближайшим объектом.
Если точка найдена, то
      {
      no = номер пересекаемого объекта.
      Вычисляем нормаль к видимой стороне поверхности объекта.
      Если (Kd > 0) то              //задано свойство диффузного отражения
            {
            Id = Интенсивность для диффузного отражения.
            }
      Если (Ks > 0) то              //зеркальные блики от источников света
            {
            Определяем направление отраженного луча dirR.
            Is = Интенсивность зеркальности.
            }
      Если (Kr > 0) то             //зеркальное отражение других объектов
            {
            Определяем направление отраженного луча dirR.
            Ir = ЛУЧ (ind+1, отраженный, dirR, no).
            }
      Если (Kt > 0) то              //объект полупрозрачный
            {
            Определяем направление преломленного луча dirT.
            It = ЛУЧ (ind+1, преломленный, dirT, no).
            }
      return Ka Ia C + Kd Id C + Ks Is + Kr Ir + Kt It .
      }
иначе
      {
      Это означает, что луч уходит в свободное пространство.
      return значение интенсивности по умолчанию.
      }
}
```

Функция ЛУЧ представлена здесь в схематичном и достаточно обобщенном виде (хотя и не в самом общем — например, здесь не предусмотрено диффузное преломление). Для правильного функционирования процесса обратной трассировки необходимо предусмотреть еще некоторые действия, кото-

рые не описаны в тексте функции. Сделайте это самостоятельно в виде упражнения. Так, например, необходимо предусмотреть, чтобы при трассировке первичного луча можно было бы получить изображение источников света, не заслоняемых другими объектами. Кроме того, когда выпускается преломленный луч от границы внутрь некоторого объекта, и далее, когда он достигнет внешней границы, то в этот момент можно заблокировать появление зеркального луча, направленного внутрь объекта (например, при моделировании стеклянной линзы, заданной в виде объема, ограниченного некоторой поверхностью). Также следует предусмотреть, чтобы зеркальный луч мог быть выпущен при трассировке преломленного луча, но только если луч пересекает иной объект, нежели тот, от границы объема которого началось преломление (например, при изображении рыбок в аквариуме). Для этого предусмотрены аргументы "тип луча" и "номер объекта". Для предотвращения "зацикливания" (например, при попадании луча внутрь пространства, ограниченного зеркальными стенками) предусмотрен аргумент "номер итерации". Этот же аргумент можно использовать для распознавания первичного луча.

Вы, наверное, уже заметили, что определение функции ЛУЧ является *рекурсивным*, то есть в теле функции присутствуют вызовы этой же функции. Так можно достаточно просто моделировать все дерево лучей: сначала первичный луч, затем — порождающиеся вторичные, третичные и так далее. Весь процесс обратной трассировки для одной точки изображения представляется в виде единственной строчки

***I* = *ЛУЧ* (1, *первичный, направление проецирования,* 0).**

Для первичного луча необходимо задать направление, соответствующее выбранной проекции. Если проекция центральная, то первичные лучи расходятся из общей точки, для параллельной проекции первичные лучи параллельны. Луч можно задать, например, координатами начальной и конечной точек отрезка, координатой начальной точки и направлением, или же еще как-нибудь. *Задание первичного луча полностью определяет проекцию изображаемой сцены.* В главе 2 мы рассматривали основные типы проекции и связанные с ними преобразования координат. А при обратной трассировке лучей какие-либо преобразования координат вообще не обязательны. Проекция получается автоматически — в том числе, не только плоская, но и, например, цилиндрическая или сферическая. Это одно из проявлений универсальности метода трассировки.

В ходе трассировки лучей необходимо определять точки пересечения прямой линии луча с объектами. Способ определения точки пересечения зависит от того, какой это объект, и каким образом он представлен в некоторой графи-

ческой системе. Так, например, для объектов, представленных в виде многогранников и полигональных сеток, можно использовать известные методы определения точки пересечения прямой и плоскости, рассматриваемые в аналитической геометрии [16]. Однако если ставится задача определения пересечения луча с гранью, то необходимо еще, чтобы найденная точка пересечения лежала бы внутри контура грани.

Известны несколько способов проверки произвольной точки на принадлежность полигону. Рассмотрим две разновидности, в сущности, одного и того же метода (рис. 4.52).

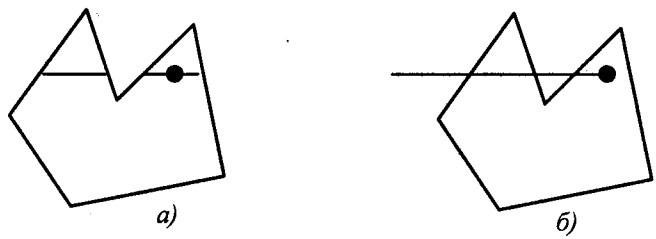


Рис. 4.52. Точка является внутренней, если:
а — точка принадлежит секущему отрезку, *б* — число пересечений нечетно

Первый способ. Находятся все точки пересечения контура горизонталью, соответствующей координате *Y* заданной точки. Точки пересечения сортируются по возрастанию значений координат *X*. Пары точек пересечения образуют отрезки. Если проверяемая точка принадлежит одному из отрезков (для этого сравниваются координаты *X* заданной точки и концов отрезков), то она является внутренней.

Второй способ. Определяется точка, лежащая на одной горизонтали с испытуемой точкой, причем требуется, чтобы она лежала вне контура полигона. Найденная внешняя точка и испытуемая являются концами горизонтального отрезка. Определяются точки пересечения данного отрезка с контуром полигона. Если количество пересечений нечетно, это значит, что испытуемая точка является внутренней [32, 33].

Если точек пересечения луча с объектами несколько, то выбирается ближайшая точка по направлению текущего луча.

В сложных сценах со многими объектами для нахождения ближайшей точки пересечения необходимо перебирать все объекты. Если каждый объект представляется многими гранями-полигонами, то нужно анализировать еще и каждую грань на предмет пересечения с лучом. Для ускорения этого процесса используется ***метод оболочек*** [28]. Суть данного метода в том, что при

переборе объектов анализируются сначала не сами объекты, а более простые формы — оболочки. Оболочка должна удовлетворять следующим требованиям. Во-первых, она должна охватывать объект, который должен целиком умещаться в ней. Если луч не пересекает оболочку, значит, этот же луч не пересечет объект. Во-вторых, процедура определения пересечения луча и оболочки должна быть как можно проще, а главное, наиболее быстрой. Использование оболочек позволяет в ходе перебора объектов сразу отбрасывать те, которые заведомо не пересекаются с лучом. Но если луч пересекает оболочку, тогда ищется точка пересечения луча с объектом. Очевидно, что если луч пересек оболочку, то не обязательно этот луч пересечет соответствующий объект — форма оболочки не совпадает с формой объекта.

В качестве оболочек можно использовать шар, параллелепипед, цилиндр и другие простые формы.

Если объектов достаточно много, то объекты (или оболочки) можно объединять в группы — для нескольких объектов одна оболочка. Таким образом, выстраивается уже иерархия оболочек: на нижнем уровне оболочки для одиночных объектов, на следующем уровне — оболочки оболочек и так далее. Такая древовидная структура может иметь несколько уровней. Это позволяет существенно ускорить процесс перебора, сделать время работы пропорциональным (теоретически) логарифму числа объектов.

Необходимо отметить, что метод оболочек можно применять не только для трассировки лучей. Этот метод является достаточно универсальным методом ускорения вычислительных процессов переборного типа. По крайней мере, в алгоритмах компьютерной графики он используется часто. Иногда метод оболочек называется по-другому, но от этого суть не меняются.

Для упрощения некоторых операций, выполняемых в ходе обратной трассировки, можно использовать следующий способ. Он разработан автором этой книги и заключается в следующем. В ходе трассировки лучей с каждым лучом связывается локальная система координат. Центр этой трехмерной декартовой системы располагается в точке, из которой направляется текущий луч трассировки. Ось Z направлена противоположно лучу, расположение осей X и Y безразлично. Таким образом, координаты X и Y точки пересечения луча с объектами всегда равны нулю. Это позволяет упростить нахождение координат точки пересечения луча, поскольку направления луча всегда одно и то же, и вычислять нужно только координату Z. Для плоских полигональных граней это можно сделать с помощью линейной интерполяции координат Z соответствующих вершин. Кроме того, упрощаются и некоторые другие операции. В частности, для отбора ближайшей точки пересечения достаточно анализировать только координату Z. Следует заметить, что упрощение

отдельных операций достигается за счет усложнения других — например, необходимо вычислять коэффициенты преобразования координат для каждой локальной системы, а также выполнять преобразования координат объектов.

В главе 7 приведен пример графической программы (studex13), реализующей данный способ, а здесь мы еще рассмотрим некоторые теоретические аспекты. На рис. 4.53 показаны локальные координаты.

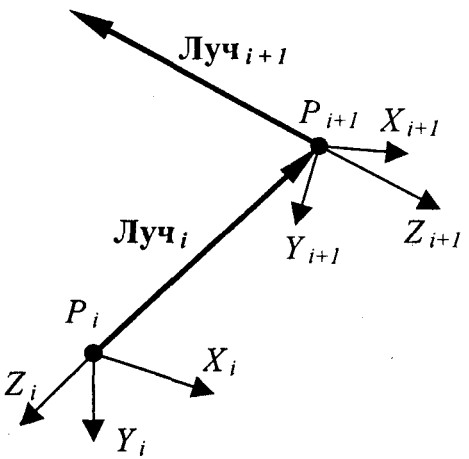


Рис. 4.53. Локальные координаты для текущих лучей

Пусть из точки P_i выпущен i-й луч, который пересек ближайший объект в точке P_{i+1}. Координаты точки P_{i+1} в системе координат (X_i, Y_i, Z_i) равны $(0, 0, z_p)$. Предположим, что направление нового, $(i+1)$-го, луча определяется в виде параллельного ему радиус-вектора в этой же системе координат, то есть (X_i, Y_i, Z_i). Обозначим координаты этого радиус-вектора через (x_r, y_r, z_r). Как определить новую систему координат, связанную с $(i+1)$-м лучом? Эту систему координат можно получить следующим образом. Вначале сдвинуть систему (X_i, Y_i, Z_i) по оси Z_i на величину z_p, а затем выполнить поворот так, чтобы ось Z_{i+1} была бы направлена против нового луча. Запишем формулы преобразований в следующем виде:

$$X_{i+1} = X_i \cos\alpha - Y_i \sin\alpha,$$
$$Y_{i+1} = X_i \sin\alpha \cos\beta + Y_i \cos\alpha \cos\beta - (Z_i - z_p) \sin\beta,$$
$$Z_{i+1} = X_i \sin\alpha \sin\beta + Y_i \cos\alpha \sin\beta + (Z_i - z_p) \cos\beta,$$

где α и β — углы поворота.

Выразим поворот не углами, а через координаты радиус-вектора, который указывает новое направление оси Z. Рассмотрим рис. 4.54. Пусть радиус-вектор имеет координаты (x, y, z). Длина радиус-вектора равна R:

$$R = \sqrt{x^2 + y^2 + z^2},$$

а длина проекция на плоскость $(X0Y)$:

$$r = \sqrt{x^2 + y^2}.$$

Тогда:

$$\cos\alpha = y / r,$$
$$\sin\alpha = x / r,$$
$$\cos\beta = z / R,$$
$$\sin\beta = r / R.$$

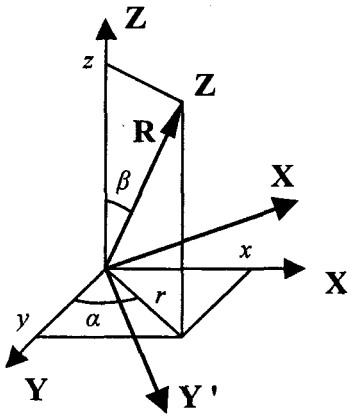


Рис. 4.54. Поворот осей

Подставим эти значения в формулы преобразования координат, учитывая, что радиус-вектор нового трассируемого луча направлен противоположно направлению оси Z_{i+1}, то есть $(x, y, z) = (-x_r, -y_r, -z_r)$. Запишем формулы в следующем виде:

$$X_{i+1} = -X_i (y_r / r) + Y_i (x_r / r),$$
$$Y_{i+1} = X_i (x_r / r)(z_r / R) + Y_i (y_r / r)(z / R) - Z_i (r / R) + z_p (r / R),$$
$$Z_{i+1} = -X_i (x_r / R) - Y_i (y_r / R) - Z_i (z_r / R) + z_p (z_r / R).$$

В матричной форме

$$\begin{bmatrix} X_{i+1} \\ Y_{i+1} \\ Z_{i+1} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{i+1} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}.$$

Матрица $\mathbf{A}_{i+1}$ соответствует аффинному преобразованию сдвига и поворота. В ходе трассировки лучей нам необходимо выполнять преобразования из мировой системы в систему координат текущего луча. Обозначим матрицу преобразований через $\mathbf{M}_{i+1}$:

$$\begin{bmatrix} X_{i+1} \\ Y_{i+1} \\ Z_{i+1} \\ 1 \end{bmatrix} = \left[\mathbf{M}_{i+1} \right] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

Поскольку $\mathbf{M}_1 = \mathbf{A}_1$, то

$$\left[\mathbf{M}_{i+1} \right] = \left[\mathbf{A}_{i+1} \right] \left[\mathbf{M}_i \right].$$

Таким образом, матрица для системы координат нового луча образуется умножением матрицы предыдущего луча на матрицу сдвига и поворота.

Важным моментом при расчетах отражения и преломления является определение нормали к видимой для текущего трассируемого луча стороны участка поверхности (рис. 4.55).

Введение локальной системы координат значительно упрощает такой выбор направления нормали. Можно руководствоваться следующим правилом — среди двух противоположно направленных векторов нормали (N и $-N$) выбирается тот, у которого соответствующий радиус-вектор в локальной системе координат (X_i, Y_i, Z_i) имеет положительную координату Z.

Рассмотрим вычисление вектора зеркально отраженного луча (**Луч** $_{i+1}$). Для этого воспользуемся уже известной нам векторной формулой, которую здесь запишем для единичных векторов: $\boldsymbol{R} = 2\,\boldsymbol{N}\,(\boldsymbol{N} \cdot \boldsymbol{S}) - \boldsymbol{S}$. Вычисляемый по этой формуле вектор $\boldsymbol{R}$ направлен так же, как искомый вектор **Луч** $_{i+1}$, а единичный вектор $\boldsymbol{S}$ нацелен противоположно вектору падающего луча, то есть вектору **Луч** $_i$.

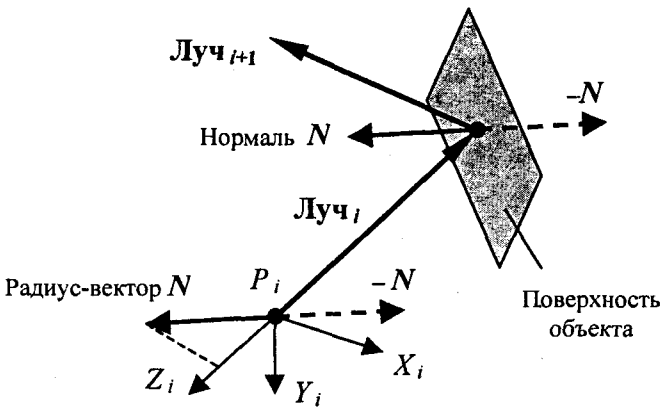


Рис. 4.55. К выбору нормали видимой стороны объекта

Еще раз подчеркнем, что речь здесь идет о лучах обратной трассировки, для которой падающий и отраженный лучи имеют противоположный смысл световым лучам, распространяемым в реальном случае.

Координаты вектора $\boldsymbol{S}$ составляют (0, 0, 1) в локальной системе. Обозначим координаты единичного вектора нормали $\boldsymbol{N}$ через (x_N, y_N, z_N). Найдем вначале скалярное произведение $(\boldsymbol{N} \cdot \boldsymbol{S}) = (0 \cdot x_N + 0 \cdot y_N + 1 \cdot z_N) = z_N$. Координаты вектора отраженного луча ($\boldsymbol{R}$) составляют

$$x_R = 2\, x_N\, z_N,$$
$$y_R = 2\, y_N\, z_N,$$
$$z_R = 2\, z_N\, z_N - 1.$$

Очевидно, что направление вектора не изменится, если все его координаты разделить на два. Тогда можно записать для координат радиус-вектора отраженного луча ($\textbf{Луч}_{i+1} = 0.5\, \boldsymbol{R}$) такие формулы:

$$x = x_N\, z_N,$$
$$y = y_N\, z_N,$$
$$z = z_N\, z_N - 0.5.$$

Теперь займемся вычислением координат преломленного луча. Для этого воспользуемся формулой идеального (зеркального) преломления:

$$\boldsymbol{T} = n\boldsymbol{N}(\boldsymbol{S} \cdot \boldsymbol{N}) - n\boldsymbol{S} - \boldsymbol{N}\sqrt{1 + n^2\left((\boldsymbol{S} \cdot \boldsymbol{N})^2 - 1\right)}.$$

Так как вектор $\boldsymbol{S} = (0, 0, 1)$, то

$$x_T = n \cdot x_N z_N - x_N\sqrt{1 + n^2\left(z_N^2 - 1\right)},$$
$$y_T = n \cdot y_N z_N - y_N\sqrt{1 + n^2\left(z_N^2 - 1\right)},$$
$$z_T = n \cdot z_N^2 - n - z_N\sqrt{1 + n^2\left(z_N^2 - 1\right)}.$$

Если подкоренное выражение отрицательно, то преломленный луч не может быть построен.

Использование локальной системы координат, ось Z которой совпадает с линией текущего трассируемого луча, позволяет упростить также определение пересечения с оболочкой объекта. Наиболее просто определяется пересече-

ние луча с оболочкой-шаром. Поскольку в любой локальной системе координат, задаваемой линейными преобразованиями, проекция шара есть круг, то достаточно узнать локальные координаты центра круга (Xc и Yc) и проверить, выполняется ли следующее соотношение: $(Xc)^2 + (Yc)^2 \le (\textit{радиус оболочки})^2$. Если выполняется, то текущий луч пересекает оболочку.

В качестве оболочки можно использовать и другую форму, например цилиндр или параллелепипед. Поскольку определение пересечения луча с произвольно повернутым цилиндром или параллелепипедом уже не столь простая задача, то здесь можно использовать следующее. Хотя расположение осей координат X, Y локальной системы ранее было нам безразлично (главное, чтобы ось Z лежала бы на линии луча), но, оказывается, приведенные выше формулы преобразования координат для локальной системы обеспечивают сохранение одинаковой ориентации для вертикальных линий. Ранее мы уже отмечали это свойство выбранного способа поворота осей применительно к аксонометрической проекции. Поэтому возможно определить проекции оболочки для всех поворотов локальных систем координат в виде прямоугольника некоторых размеров в плоскости ($X0Y$). Определение пересечения луча с оболочкой сводится к проверке, находится ли точка локальных координат $(X_i, Y_i) = (0, 0)$ внутри прямоугольника, описывающего оболочку.

А теперь сделаем общие выводы по методу обратной трассировки лучей. Положительные черты:

1. Универсальность метода, его применимость для синтеза изображений достаточно сложных пространственных схем. Воплощает многие законы геометрической оптики. Просто реализуются разнообразные проекции.

2. Даже усеченные варианты данного метода позволяют получить достаточно реалистичные изображения. Например, если ограничиться только первичными лучами (из точки проецирования), то это дает удаление невидимых точек. Трассировка уже одного-двух вторичных лучей дает тени, зеркальность, прозрачность.

3. Все преобразования координат (если таковые есть) линейны, поэтому достаточно просто работать с текстурами.

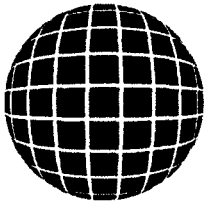
4. Для одного пиксела растрового изображения можно трассировать несколько близко расположенных лучей, а потом усреднять их цвет для устранения эффекта ступенчатости (антиалиасинг).

5. Поскольку расчет отдельной точки изображения выполняется независимо от других точек, то это может быть эффективно использовано при реали-

зации данного метода в параллельных вычислительных системах, в которых лучи могут трассироваться одновременно.

Недостатки:

1. Проблемы с моделированием диффузного отражения и преломления.
2. Для каждой точки изображения необходимо выполнять много вычислительных операций. Трассировка лучей относится к числу самых медленных алгоритмов синтеза изображений.



ГЛАВА 5

Примеры изображения трехмерных объектов

5.1. Шар

Рассмотрим несколько способов построения изображения шара.

Каркасное изображение

Для каркасного изображения шара можно рисовать сетку меридианов и параллелей. Для этого удобно воспользоваться известными формулами параметрического описания. Координаты точек поверхности шара определяются как функции от двух переменных (параметров) — широты и долготы (рис. 5.1):

$$x = R \cos B \sin L,$$
$$y = R \cos B \cos L,$$
$$z = R \sin B,$$

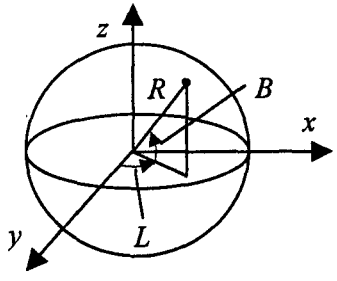


Рис. 5.1. Широта и долгота

где R — радиус шара, B — широта (изменяется от –90° до +90°), L — долгота (от –180° до +180° или от 0° до 360°).

Меридиан — это линия, представляющая точки с постоянной долготой. Каркас из меридианов можно нарисовать следующим образом:

```
for (L=0; L<360; L1+=dL)
   for (B=-90; B<=90; B+=dB)
        {
         x = R cos B sin L ;
```

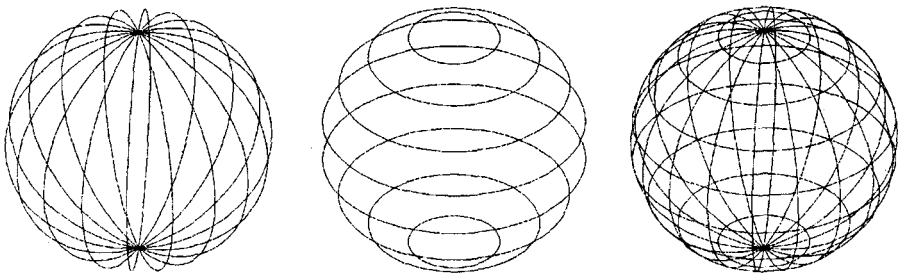
```
 y = R cos B cos L ;
 z = R sin B ;
(X, Y) = Преобразование координат (x, y, z);
 Рисование отрезка до точки (X, Y);
}
```

Здесь введены две величины — *dL* и *dB*. Значение *dL* определяет шаг меридианов по долготе, значение *dB* — это шаг по широте, который должен быть малым (единицы градусов) для изображения меридиана достаточно гладкой кривой. Преобразование координат производится по формулам, соответствующим выбранной проекции. Для аксонометрической проекции — это поворот мировых координат на углы α и β, с последующим преобразованием видовых координат в экранные координаты (как было рассмотрено в главе 2).

Параллель — это линия, состоящая из точек с постоянной широтой. Для рисования каркаса из параллелей можно использовать такой цикл:

```
for (B=-90; B<=90; B+=dB)
   for (L=0; L<=360; L1+=dL)
        {
         x = R cos B sin L ;
         y = R cos B cos L ;
         z = R sin B ;
         (X, Y) = Преобразование координат (x, y, z);
         Рисование отрезка до точки (X, Y);
        }
```

На рис. 5.2 изображены примеры каркасного изображения на основе меридианов и параллелей.



Меридианы $dL = 20°$ | Параллели $dB = 20°$ | $dB = dL = 20°$

Рис. 5.2. Меридианы и параллели

Удаление невидимых точек

Для изображения поверхности шара с удалением невидимых точек в аксонометрической проекции можно воспользоваться таким свойством: видимыми являются точки с неотрицательным значением координаты Z в системе видовых координат. При этом центр видовых координат (X, Y, Z) совпадает с центром шара, плоскость $X0Y$ является плоскостью проецирования, а ось Z направлена на камеру (наблюдателя). На рис. 5.3 приведен простейший вариант показа поверхности шара с удалением невидимых точек меридианов и параллелей.

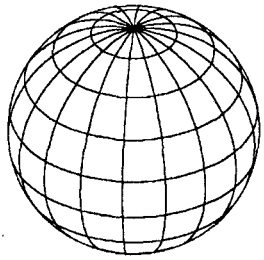


Рис. 5.3. Удалены невидимые точки

Приведенные выше изображения были построены на основе линий и являются схематичными, весьма далекими от реалистичного изображения поверхности шара. Значительно лучше можно нарисовать шар с помощью фигур с заполнением — закрашиванием поверхности разными цветами.

Многогранник с закрашиванием граней

Теперь нарисуем шар в виде многогранника, аппроксимирующего форму поверхности с заданной точностью. Известно большое число типов многогранников. Сведения о них имеются в многочисленных книгах по геометрии.

Мы будем рассматривать достаточно узкий класс многогранников. Отличительным признаком рассматриваемых многогранников является то, что у них ребра ориентированы вдоль меридианов и параллелей. Такие многогранники будем описывать двумя параметрами — шаг по широте (dB) и шаг по долготе (dL) в градусах. Кроме того, будем считать эти многогранники вписанными изнутри в шар, то есть вершины каждой грани лежат на поверхности шара. Очевидно, что с увеличением количества граней такие многогранники все больше приближаются к шару. Иными словами, чем меньше dB и dL, тем лучше поверхность многогранника аппроксимирует форму поверхности шара.

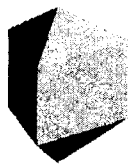
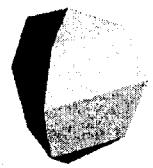
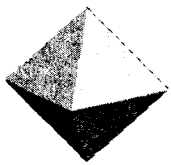
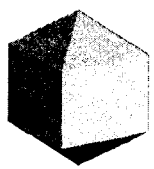
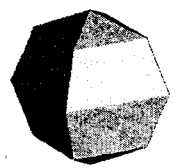
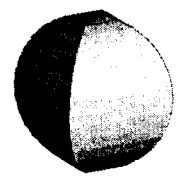
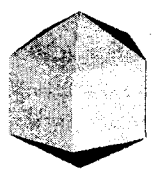
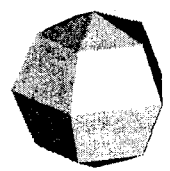
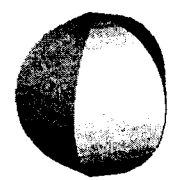
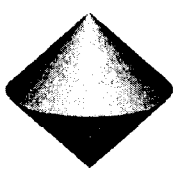
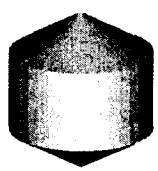
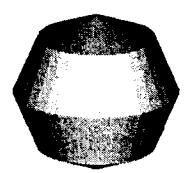
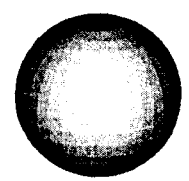
В табл. 5.1 приведены примеры таких многогранников.

Некоторые из этих многогранников достаточно интересны по форме, а многогранник с параметрами $dB = 90°$, $dL = 90°$ называется **октаэдром**.

Все грани, примыкающие к верхнему и нижнему полюсам, являются треугольными, а остальные грани — четырехугольными. С позиций компьютерной графики это выгодно отличает многогранники рассматриваемого типа

от других, например, правильных многогранников, у которых грани могут иметь значительно больше ребер, чем четыре.

Таблица 5.1

dL \ dB	90°	60°	45°	10°
120°				
90°				
72°				
10°				

Для построения изображения многогранников будем рисовать каждую грань как полигон. Для вывода полигона требуется определить координаты всех его вершин и задать цвет заполнения.

Очевидно, что показ граней одним цветом дает неудовлетворительный результат — полностью теряется форма поверхности трехмерного объекта. Показ ребер (контуров полигонов граней) другим цветом немного улучшает восприятие (рис. 5.4).

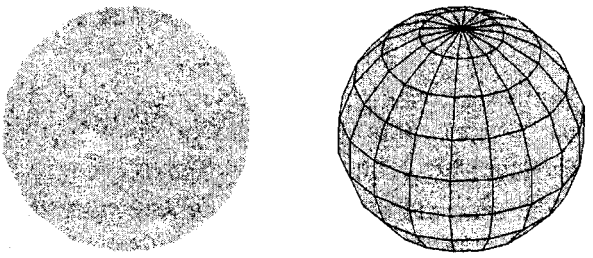


Рис. 5.4. Неудачное изображение — все грани одного цвета

Построим изображение так, чтобы грани были окрашены в соответствии с законами отражения света. Общую схему алгоритма можно изобразить таким образом:

```
for (B=-90; B<90; B+=dB)
   for (L=0; L<360; Ll+=dL)
        {
         x[0] = R cos B sin L ;
         y[0] = R cos B cos L ;
         z[0] = R sin B ;

         x[1] = R cos (B+dB) sin L ;
         y[1] = R cos (B+dB) cos L ;
         z[1] = R sin (B+dB) ;

         x[2] = R cos (B+dB) sin (L+dL) ;
         y[2] = R cos (B+dB) cos (L+dL) ;
         z[2] = R sin (B+dB) ;

         x[3] = R cos B sin (L+dL) ;
         y[3] = R cos B cos (L+dL) ;
         z[3] = R sin B ;
         (X[i], Y[i], Z[i]) = Поворот координат (x[i], y[i], z[i]) ;
        Определение вектора нормали по (X[i], Y[i], Z[i]) и определение
цвета грани, учитывающего ее ориентацию относительно источника света.
        Рисование выбранным цветом полигона грани с четырьмя вершинами
(X[i], Y[i], Z[i]);
        }
```

Приведенную тут схему алгоритма не следует воспринимать как программу на языке С, а также сам алгоритм как оптимальный. В частности, здесь много лишних вычислений косинусов и синусов — количество таких операций обязательно нужно уменьшать, поскольку они выполняются в традиционных компьютерах очень медленно. Кроме того, здесь и далее мы будем рисовать все грани четырехугольниками, хотя, лишь незначительно усложнив данную

схему, можно предусмотреть и отдельную ветвь для треугольных граней у полюсов.

Для удаления невидимых точек можно использовать Z-буфер, но можно перестроить цикл так, чтобы грани рисовались в таком порядке — начиная с самых дальних и заканчивая ближайшими.

Для определения цвета каждой грани нужно учесть взаимное расположение источника света и нормали грани. Как уже было показано выше для диффузной и зеркальной моделей отражения, важно определить координаты вектора нормали. Мы уже рассматривали общий способ вычисления координат нормали к произвольной плоской грани (см. разд. 4.3). Этот способ, на основе векторного произведения, можно применить, естественно, и в данном случае. Однако для таких многогранников как наши, нормали к граням можно вычислять значительно проще и быстрее. Поскольку радиус-вектор для каждой точки поверхности шара и есть вектор нормали, то координаты каждой точки поверхности шара являются координатами вектора нормали, если координаты центра шара равны (0, 0, 0). Таким образом, для вычисления координат вектора нормали к грани достаточно определить координаты точки поверхности шара, соответствующей центру грани. Или же можно взять сумму координат вершин грани и разделить на число вершин — это уже приближенный неточный способ.

На рис. 5.5 изображены многогранники с различным числом граней, причем источник освещения расположен сзади нас — на оси Z видовых координат.

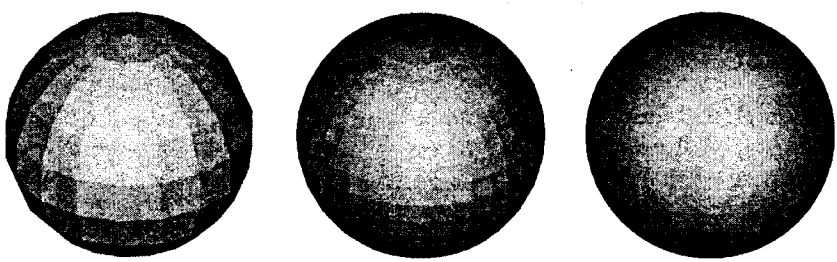


Рис. 5.5. Закрашивание граней с учетом их ориентации

Как видим, для создания иллюзии гладкой поверхности шара нужно очень большое число граней (очевидно, что это число возрастает с увеличением видимого радиуса шара). Если рассматривать рис. 5.5, то при $dB = dL = 5°$ многогранник уже похож на шар — отдельные грани достаточно сложно различить на типографском отпечатке этой книги (здесь также сказывается то, что при печати используется дизеринг, который сглаживает изображение).

Однако на экране дисплея грани все еще заметны. Глаз человека подчеркивает границы.

Закрашивание граней методом Гуро

Метод Гуро дает хорошие результаты для создания иллюзии гладкой поверхности шара, изображаемого достаточно большим числом плоских граней. В соответствии с этим методом для каждой грани нужно определить векторы в вершинах, затем для каждой вершины вычислить значение интенсивности отраженного света и потом, при выводе полигона грани, интерполировать интенсивности вершин для вычисления интенсивности отраженного света в каждой точке внутри грани.

Значительным удобством при вычислении координат нормалей в вершинах граней в нашем случае является то, что координаты вектора нормали в любой вершине грани совпадают с координатами самих вершин — если центр шара имеет координаты (0, 0, 0).

Рассмотрим случай, когда точечный источник света расположен на оси Z видовых координат. Используем диффузную модель отражения. В этом случае рассчитать интенсивность отраженного света в вершине грани можно следующим образом. Косинус угла между осью Z и нормалью

$$\cos\theta = \frac{Zn}{\sqrt{Xn^2 + Yn^2 + Zn^2}},$$

где *Xn*, *Yn* и *Zn* — видовые координаты вектора нормали.

Поскольку координаты вектора нормали в вершине равны координатам вершин, а сумма квадратов координат (в том числе и видовых) для любой точки поверхности шара равна квадрату его радиуса, то

$$\cos\theta = \frac{Zv}{R},$$

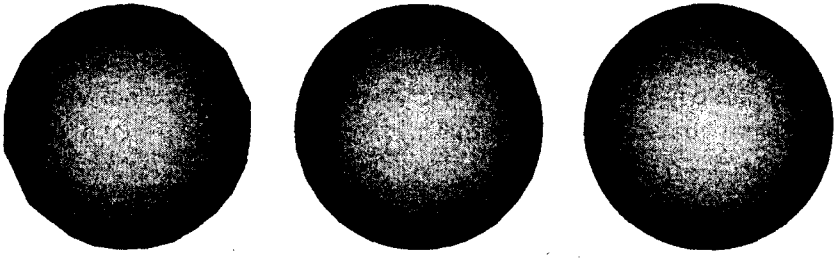
где *Zv* — видовая координата вершины грани.

Алгоритм вывода многогранника для метода Гуро аналогичен рассмотренному выше алгоритму. Запишем его следующим образом:

```
for (B=-90; B<90; B+=dB)
  for (L=0; L<360; L1+=dL)
    {
     Определение мировых координат вершин грани x[i], y[i], z[i].
```

```
    Поворот координат x[i], y[i], z[i] для определения видовых
координат X[i], Y[i], Z[i].
    Вычисление интенсивности отраженного света в вершинах. В простейшем
случае для диффузного рассеивания  Iотр[i] = Iист * Z[i] / R.
    Преобразование видовых координат в координаты устройства
графического вывода (экранные координаты).
    Рисование полигона с интерполяцией интенсивностей.
    }
```

На рис. 5.6 представлены многогранники с различным числом граней, закрашенные в соответствии с методом Гуро.



dB = *dL* = 20° *dB* = *dL* = 10° *dB* = *dL* = 5°

Рис. 5.6. Закрашивание Гуро

Как видно на рис. 5.6, иллюзия гладкости поверхности достигается уже при достаточно малом числе граней. Так, например, при *dB* = *dL* = 20° в большей степени заметны неровности контура, чем погрешности закрашивания.

Учет расположения источника света

В приведенных выше примерах использовался точечный источник освещения, расположенный так же, как и камера — на оси Z видовых координат. Как получить изображение шара, освещенного сбоку?

Рассмотрим для простоты диффузное рассеивание. Для расчета косинуса угла наклона нормали по отношению к направлению луча источника света удобно ввести еще одну систему координат. Эта система координат — обозначим ее как (*Xc, Yc, Zc*) — повернута в пространстве таким образом, чтобы источник света располагался на оси *Zc*. Здесь следует уточнить, что мы рассматриваем точечный источник света, расположенный снаружи шара на достаточно большом расстоянии, причем не учитываем зависимость освещенности точек поверхности от расстояния до источника света.

Ориентацию системы координат источника света (*Xc, Yc, Zc*) можно задать двумя углами поворота (α_c, β_c) — подобно тому, как задается видовая система координат. На рис. 5.7 показано расположение осей.

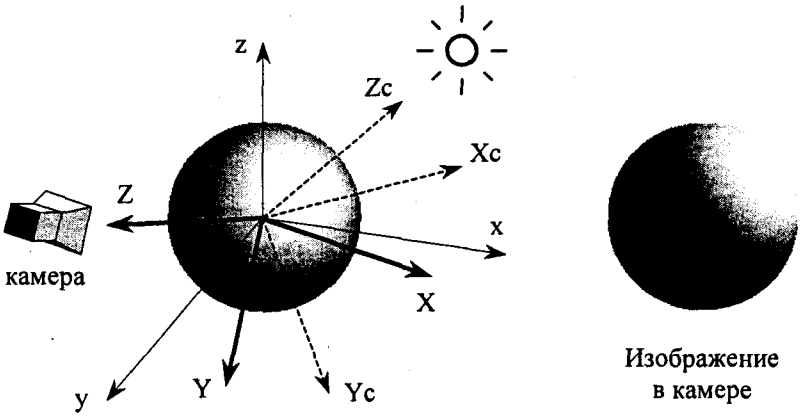


Рис. 5.7. Пример взаимного расположения осей мировых координат (*x, y, z*), видовых координат (*X, Y, Z*) — ($\alpha = -18°$, $\beta = 71°$) и системы координат, связанной с источником освещения (*Xc, Yc, Zc*) — ($\alpha_c = 51°$, $\beta_c = 48°$)

Косинус угла нормали можно вычислить по формуле

$$\cos\theta = \frac{Zc}{R},$$

где *Zc* — координата точки поверхности в системе координат источника света. Необходимо учесть, что нулевые и отрицательные значения косинуса соответствуют неосвещенным точкам поверхности.

Таким образом, для рисования поверхности с учетом произвольной ориентации источника света необходимо определять еще и значение координаты *Zc*. В остальном алгоритм рисования шара полностью аналогичен уже рассмотренным алгоритмам.

Градиентное закрашивание круга

В предыдущем разделе мы пытались изобразить гладкую поверхность шара путем рисования многогранника с достаточно сложным закрашиванием полигонов отдельных граней. А нельзя ли рисовать шар как-нибудь попроще?

Если рассматривать общий контур шара — то это линия круга. Остается лишь каким-то образом закрасить внутренности круга, чтобы цвет пикселов более или менее соответствовал бы изображению объемного объекта. Нужно закрашивать так, чтобы цвет пикселов внутри круга плавно изменялся бы в соответствии с освещенностью поверхности шара. Для этого можно использовать *градиентное* закрашивание.

Для простоты будем полагать, что источник света расположен позади нас. В этом случае можно использовать радиальное градиентное закрашивание — наиболее яркие точки расположены в центре, яркость остальных точек убывает пропорционально расстоянию до центра круга.

Вашему вниманию предлагается следующий алгоритм закрашивания.

```
R2 = R*R;                    //квадрат радиуса шара
for (x=0; x<=R; x++)
  for (y=0; y<=x; y++)
    {
    r2 = x*x + y*y;           //квадрат расстояния до центра круга
    if (r2 > R2) break;       //вышли за пределы круга, прерываем цикл
    k = 1 - r2 / R2;          //коэффициент отражения k = от 0 до 1
    Цвет пикселов (R, G, B) = (k*Rmax, k*Gmax, k*Bmax);
    Рисуем пикселы во всех октантах:
       Пиксел (xc+x, yc+y);   //xc и yc - координаты центра
       Пиксел (xc+x, yc-y);
       Пиксел (xc-x, yc+y);
       Пиксел (xc-x, yc-y);
       Пиксел (xc+y, yc+x);
       Пиксел (xc+y, yc-x);
       Пиксел (xc-y, yc+x);
       Пиксел (xc-y, yc-x);
    }
```

В этом алгоритме использована симметрия шара, освещенного указанным образом. Поэтому вычисления производятся только для точек одного октанта круга. Для определения цвета пикселов вычисляется значение (k), которое равно квадрату косинуса угла нормали по отношению к оси Z видовых координат.

Во второй части этой книги мы рассмотрим примеры программ, в которых использован данный алгоритм градиентного закрашивания.

Проанализируем алгоритм и попробуем оптимизировать его по быстродействию. В первую очередь отметим, что в данном алгоритме вычисляется квадрат косинуса нормали именно из соображений повышения быстродейст-

вия — для вычисления косинуса в первой степени необходимо вычислять квадратный корень из (k). Далее заметим, что во внутреннем цикле присутствуют операции деления и умножения. Как от них избавиться? Вначале рассмотрим вычисление величины $r2 = x^2+y^2$. Поскольку во внутреннем цикле изменяется y, то рассмотрим $r2$ как функцию от y. Обозначим ее как $r2(y)$, и найдем разность значений этой функции для соседних шагов цикла:

$$\Delta'(y+1) = r2(y+1) - r2(y) = x^2 + (y+1)^2 - x^2 - y^2 = 2y+1 .$$

Поскольку $r2(y+1) = r2(y) + \Delta'(y+1)$, то можно организовать вычисление $r2$ в цикле по y следующим образом:

```
r2 = x*x;
for (y=0; y<=x; y++)
   {
   . . .                    // другие операции в цикле

   r2 = r2 + 2*y + 1;       // r2(y+1) = r2(y) + Δ'(y+1)
   }
```

Умножение на 2 компьютер может выполнять как сдвиг двоичного числа, что несколько быстрее обычного умножения. Попробуем еще улучшить цикл. Для этого рассмотрим разности второго порядка:

$$\Delta''(y+1) = \Delta'(y+1) - \Delta'(y) = 2y+1 - 2y + 1 = 2.$$

То, что вторая разность является константой, не зависящей от y, можно использовать для построения цикла, в теле которого присутствуют только операции сложения. Перестроим цикл следующим образом.

```
r2 = x*x;
dr2 = 1;
for (y=0; y<=x; y++)
   {
   . . .                    // другие операции в цикле

   r2 = r2 + dr2;             // r2(y+1) = r2(y) + Δ'(y+1)
   dr2 = dr2 + 2;             // Δ'(y+1) = Δ'(y) + 2
   }
```

Можно применить этот разностный метод (известный в математической литературе для организации итерационных вычислений) также и для вычисления величины $k = 1 - r2/R2$. Вы можете это сделать в качестве упражнения.

Необходимо отметить, что подобная оптимизация может не дать существенного ускорения графического вывода, если время вычисления величин r^2 и k мало по сравнению с другими операциями. Так, например, в первую очередь необходимо обеспечивать быструю запись в память пикселов растра отображения. Хорошие возможности для ускорения тут дает симметрия круга по октантам.

Кроме иллюстрации возможностей ускорения, модифицированный алгоритм здесь приведен еще с одной целью. Оказывается, если заменить операцию ($dr2 = dr2 + 2$) на ($dr2 = dr2 + D$), где D является константой для всего цикла рисования, то можно получить довольно любопытные вариации формы круга. На рис. 5.8 приведена форма фигур, соответствующих некоторым значениям D.

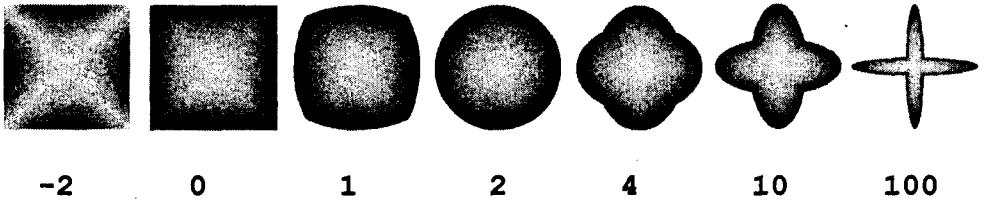


Рис. 5.8. Фигуры, получаемые при различных значениях D

Значение константы D может быть и дробным — форма фигуры плавно изменяется в зависимости от D.

Наложение текстуры на шар

До сих пор мы с вами рассматривали одноцветный шар. Однако, возможно, требуется построить изображение шара, раскрашенного более сложным образом. Часто бывает необходимо имитировать различные материалы, сложную структуру поверхности — например, пористую, с трещинами и тому подобное. Для этого могут быть использованы текстуры. Изображение текстуры как бы накладывается на поверхность объекта.

Давайте построим изображение поверхности шара, на который как резиновая оболочка "натянута" текстура — карта мира.

Из *картографии* известны многие способы изображения Земли в соответствии с различными проекциями. Выберем вертикальную цилиндрическую равнопромежуточную проекцию шара, для которой наиболее просто пересчитываются координаты — из угловых координат (широты, долготы) в пикселные координаты текстуры (x_T, y_T) и обратно (рис. 5.9).

$$x_T = \frac{X\max}{360} L + \frac{X\max}{2}$$

$$y_T = -\frac{Y\max}{180} B + \frac{Y\max}{2}$$

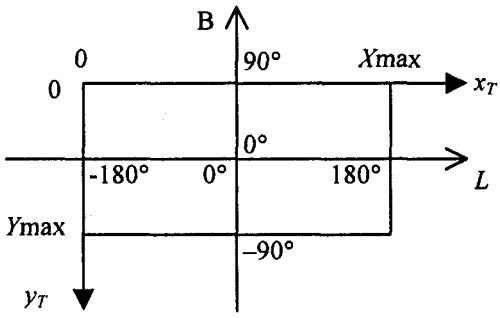


Рис. 5.9. Соотношение угловых координат поверхности шара и пикселных координат растрового прямоугольника текстуры

На рис. 5.10 приведен один из возможных вариантов карты мира в цилиндрической равнопромежуточной проекции.

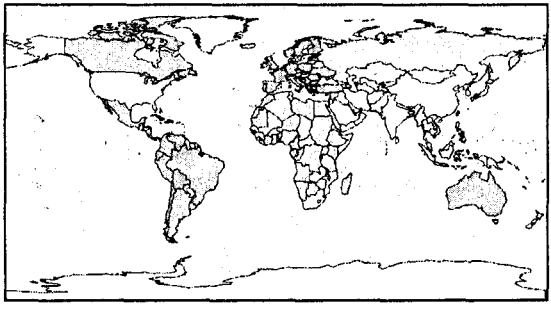


Рис. 5.10. Пример текстуры — карта мира

Как использовать текстуру для построения требуемого изображения шара? Это можно сделать несколькими способами. Разнообразие вариантов обуславливается, в том числе, и способом описания поверхности шара.

Рассмотрим вначале вариант, основывающийся на аналитической форме описания шара. Будем использовать формулу, связывающую координаты каждой точки поверхности шара в виде

$$X^2 + Y^2 + Z^2 = R^2.$$

Построение растрового изображения шара в аксонометрической проекции представим как двумерный цикл перебора всех пикселов квадрата, охватывающего круг на плоскости проецирования. При этом для каждого пиксела вначале пересчитываются координаты проекции в мировые координаты, а далее — в координаты точки в текстуре.

```
for (Y= -R; Y<=R; Y++)
    for (X= -R; X<=R; X++)
        Если (R^2 - X^2 - Y^2) больше или равно 0, то:
      {
      Z = корень квадратный из (R^2 - X^2 - Y^2);
      Преобразование координат (X, Y, Z) в плоскости проецирования
в исходные прямоугольные координаты (x, y, z);
      Преобразование координат (x, y, z) в угловые координаты (B, L);
      По значениям широты и долготы (B, L) вычисляем пикселные
координаты в растре текстуры (xТ, yТ);
      Определяем цвет пиксела в растре текстуры (тексела)
с координатами (xТ, yТ);
      Преобразование цвета тексела с учетом модели отражения света
для текущей точки поверхности шара;
      Выводим пиксел найденного цвета в растр с координатами (X, Y);
      }
```

Рассмотрим данный алгоритм подробнее. Для простоты будем использовать аксонометрическую проекцию, которая строится как поворот исходной (мировой) системы координат (x, y, z) на углы α и β, соответствующие положению камеры наблюдения. Поскольку в обеих системах координат — как в исходной (x, y, z), так и в повернутой системе (X, Y, Z) — для шара сохраняется равенство суммы квадратов координат квадрату его радиуса, то можно найти Z по известным двум координатам X и Y следующим образом:

$$Z = \pm\sqrt{R^2 - X^2 - Y^2}.$$

Поскольку мы строим изображение для передней стороны шара, то нужно брать значение Z со знаком "+". Вычисление координаты Z является ключевым моментом рассматриваемого алгоритма. Это достаточно уникальный случай простого определения трехмерных координат точек объекта по двумерным координатам проекции.

Далее, координаты точки (x, y, z) можно вычислить, выполнив обратный поворот координат (X, Y, Z). Например, если считать, что плоскость проецирования повернута вокруг оси x на угол β, то

$$x = X;$$
$$y = Y\cos\beta + Z\sin\beta;$$
$$z = -Y\sin\beta + Z\cos\beta.$$

Поворот камеры вокруг оси z (по долготе на угол α) мы учтем позже.

Преобразование координат (x, y, z) в широту и долготу (B, L) можно выполнить по формулам:

$$B = \arcsin(z/R);$$

$$L' = \begin{cases} \operatorname{arctg}(x/y), & \text{если } x > 0 \text{ и } y > 0 \quad (\textit{первый квадрант}); \\ \pi/2 + \operatorname{arctg}(-y/x), & \text{если } x > 0 \text{ и } y < 0; \\ \pi + \operatorname{arctg}(x/y), & \text{если } x < 0 \text{ и } y < 0; \\ 3\pi/2 + \operatorname{arctg}(-y/x), & \text{если } x < 0 \text{ и } y > 0. \end{cases}$$

Теперь можно учесть поворот камеры по долготе:

$$L = L' + \alpha.$$

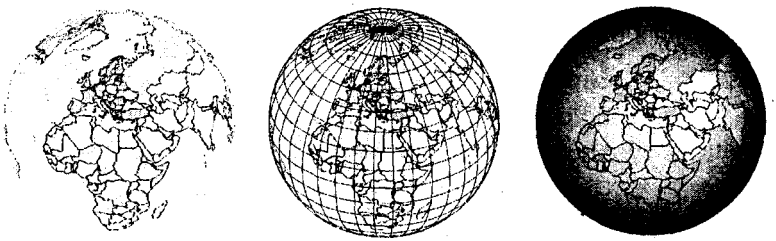
Приведенные выше формулы справедливы для случая, когда центр координат плоскости проецирования совпадает с точкой (0, 0) основного растра. При построении изображения в окне необходимо еще вычислять экранные координаты, что можно сделать путем масштабирования и сдвига осей координат.

По известным значениям широты (B) и долготы (L) определяем координаты точки в растре текстуры. Как уже указывалось выше, для текстуры в виде вертикальной равнопромежуточной цилиндрической проекции шара пересчет координат выполняется простейшим образом:

$$x_T = aL + b;$$
$$y_T = cB + d,$$

где a, b, c и d — константы, определяемые размерами растра текстуры.

На рис. 5.11 показаны несколько текстурированных изображений поверхности шара.



Без учета отражения света теряется объемность

Наложены меридианы и параллели

Диффузная модель отражения

Рис. 5.11. Примеры использования текстуры для шара

Данный способ построения текстурированного изображения имеет свои достоинства и недостатки. Положительной чертой способа можно считать высокую точность отображения — точность определяется разрешением растров текстуры и основного растра. Погрешность вычисления координат по формулам преобразований здесь незначительна при использовании стандартной разрядности чисел с плавающей точкой.

Недостатком рассмотренного способа является низкая скорость построения изображения, поскольку для каждого пиксела основного растра нужно выполнять несколько вычислительных операций, требующих много машинного времени — в первую очередь, это арксинус и арктангенс.

Разумеется, приведенный выше алгоритм можно усовершенствовать с целью повышения быстродействия. Так, например, можно использовать симметрию шара, что позволяет совместить вычисления для нескольких точек поверхности одновременно. Однако все равно приходится вычислять арксинусы и арктангенсы наряду с другими операциями с плавающей точкой.

Здесь мы рассмотрели подход, основной идеей которого было получение формул преобразования координат изображения в мировые координаты и далее — в текстурные. Формулы преобразования здесь достаточно просты с точки зрения математики, но не слишком удобны для быстрых вычислений. Необходимо отметить, что такие формулы мы получили для, возможно, простейшего объемного объекта — шара. Для других типов поверхностей, которые описываются значительно более сложными формулами, вероятно, и не существует решение в точном аналитическом виде такой задачи, как пересчет координат плоскости проецирования в мировые трехмерные. Сложность решения обуславливается сложностью формул аналитического описания поверхности. И не только. Например, для эллипсоида формулы описания поверхности не намного сложнее, чем для шара. Однако определение значения координаты Z по известным X и Y является гораздо более сложной задачей, поскольку видимые точки эллипсоида могут иметь как положительные, так и отрицательные значения координаты Z.

Значительно более универсальным и, как правило, более эффективным по быстродействию можно считать метод текстурирования, использующий представление поверхностей полигональными гранями.

Наложение текстуры на многогранник

Рассмотрим наложение растровой текстуры на шар, аппроксимированный многогранником. В главе 3 мы с вами уже обсуждали способы вывода полигонов с наложением текстуры. Эти способы были представлены в обобщен-

ном виде. Здесь мы разберемся с некоторыми особенностями наложения текстуры для шара.

Пусть в качестве текстуры будет карта, которую мы уже рассматривали выше. Кстати, во многих англоязычных графических пакетах программ файлы текстур так и называются "*maps*", хотя и не представляют собой продукцию картографии. Термин "карта" для текстур можно еще трактовать как "развертка".

В нашем случае каждая грань многогранника будет соответствовать одной прямоугольной клетке текстуры (рис. 5.12).

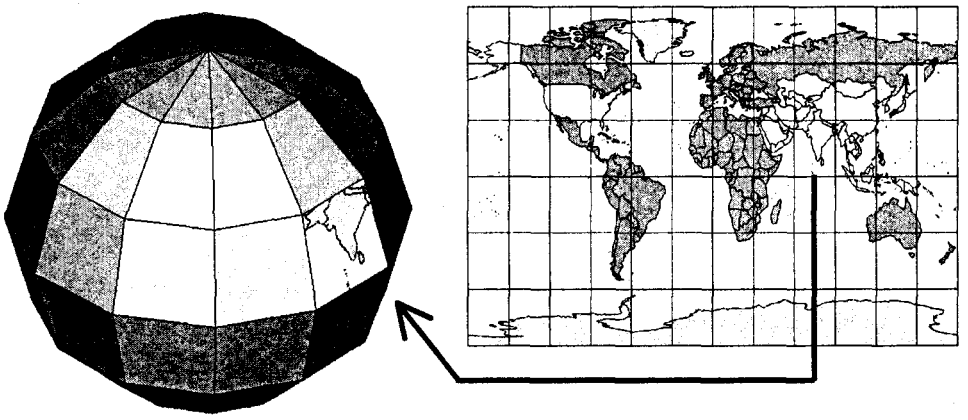


Рис. 5.12. Наложение фрагмента растра текстуры на одну грань

Для рисования каждой грани можно использовать алгоритм вывода полигонов, модифицированный следующим образом. Мы уже рассматривали выше в главе 3 один из алгоритмов вывода полигонов, который рисует полигон горизонталями заполнения внутренних точек. Операции по наложению текстуры удобно встроить в цикл рисования каждой такой горизонтали. При этом в ходе вывода пиксела горизонтали по известным текущим координатам пиксела в основном растре (X, Y) определяются соответствующие координаты (x_T, y_T) тексела в растре текстуры. Затем определяется цвет тексела, который и будет определять цвет пиксела в основном растре. Вычислить координаты (x_T, y_T) для каждой точки грани можно следующим образом. Рассмотрим рис. 5.13.

Для определения искомых координат в точке горизонтали будем использовать линейную интерполяцию координат угловых точек. При этом до начала цикла вывода горизонтали определим текстурные координаты в точках A и B — (x_{TA}, y_{TA}) и (x_{TB}, y_{TB}). Это можно сделать, разделив отрезки (0-1) и (2-3) пропорционально Y.

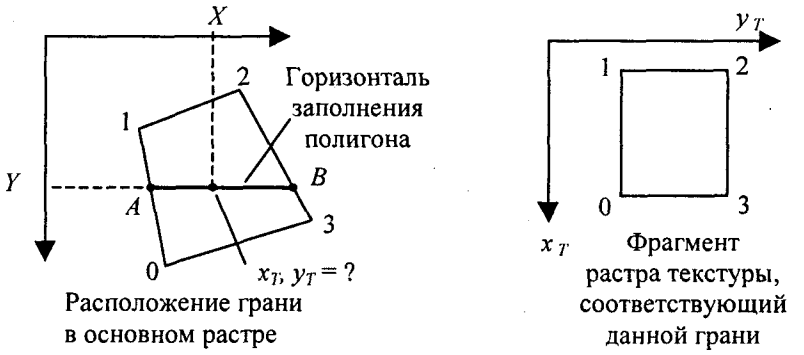


Рис. 5.13. Соответствие угловых точек грани и фрагмента текстуры

Запишем пропорции:

$$\frac{x_{TA}-x_{T1}}{x_{T0}-x_{T1}}=\frac{y_{TA}-y_{T1}}{y_{T0}-y_{T1}}=\frac{Y-Y_1}{Y_0-Y_1},$$

$$\frac{x_{TB}-x_{T2}}{x_{T3}-x_{T2}}=\frac{y_{TB}-y_{T2}}{y_{T3}-y_{T2}}=\frac{Y-Y_2}{Y_3-Y_2},$$

исходя из которых:

$$x_{TA}=x_{T1}+(x_{T0}-x_{T1})\frac{Y-Y_1}{Y_0-Y_1},$$

$$y_{TA}=y_{T1}+(y_{T0}-y_{T1})\frac{Y-Y_1}{Y_0-Y_1},$$

и

$$x_{TB}=x_{T2}+(x_{T3}-x_{T2})\frac{Y-Y_2}{Y_3-Y_2},$$

$$y_{TB}=y_{T2}+(y_{T3}-y_{T2})\frac{Y-Y_2}{Y_3-Y_2}.$$

В ходе цикла вывода горизонтали вычисление координат (x_T, y_T), соответствующих текущим пикселам горизонтали, можно выполнять делением отрезка (A-B) пропорционально X:

$$\frac{x_T-x_{TA}}{x_{TB}-x_{TA}}=\frac{y_T-y_{TA}}{y_{TB}-y_{TA}}=\frac{X-X_A}{X_B-X_A},$$

откуда

$$x_T = x_{TA} + (x_{TB} - x_{TA})\frac{X - X_A}{X_B - X_A},$$

и

$$y_T = y_{TA} + (y_{TB} - y_{TA})\frac{X - X_A}{X_B - X_A}.$$

Следует учесть, что приведенные выше формулы отражают лишь отдельный эпизод в ходе заполнения полигона грани. Очевидно, что в другие моменты времени горизонталь заполнения будет пересекать иные ребра полигона (а им будут соответствовать свои индексы вершин). Это достаточно просто реализуется в алгоритмах заполнения полигонов — в том числе и в алгоритме, приведенном в главе 3 этой книги.

Рис. 5.14. Текстура и интерполяция Гуро

Как вы, наверное, уже заметили, интерполяция координат для такой текстурированной грани аналогична интерполяции интенсивностей отраженного света для метода Гуро. Кстати, если в цикл вывода горизонтали заполнения встроить и интерполяцию по методам Гуро или Фонга, то можно получить более реалистичное текстурированное изображение — с учетом отражения света (рис. 5.14).

Необходимо принимать во внимание, что линейная интерполяция текстурных координат вершин граней (x_{Tb}, y_{Tl}) вносит погрешность. Эта погрешность значительно возрастает, если соотношение угловых координат (B, L) и (x_T, y_T) для текстуры карты мира нелинейно — а это так для многих проекций карты мира. В этом случае для уменьшения погрешности отображения целесообразно увеличивать число граней. Можно интерполировать не текстурные координаты, а широту и долготу (B, L), и для них потом в каждой точке горизонталей заполнения находить соответствующие значения (x_T, y_T) — это уже чем-то напоминает метод Фонга, не так ли?

Давайте рассмотрим еще один способ наложения текстур на многогранник. В отличие от только что рассмотренного, здесь не используется интерполяция. При интерполяции требуется выполнять достаточно много операций, в том числе операции деления. Избавиться от достаточно медленной операции деления (в общем случае — над числами с плавающей точкой) можно, если

для каждого полигона грани определить формулы преобразования координат (X, Y) в (x_T, y_T) в виде

$$x_T = aX + bY + c,$$
$$y_T = dX + eY + f,$$

причем, коэффициенты a, b, c, d, e и f являются константами в течение всего цикла вывода одной грани.

В общем случае для четырехугольной грани это невозможно.

Подобное аффинное преобразование нельзя построить по четырем точкам соответствия четырехугольников текстуры и четырехугольника грани, ориентированной произвольным образом. Преобразование формы прямоугольного фрагмента текстуры в форму четырехугольной грани многогранника значительно сложнее — попробуйте, например, выразить процесс интерполяции для рассмотренного выше способа в одной функциональной зависимости $(x_T, y_T) = F(X, Y)$ и вы увидите, что она является сложной, нелинейной.

Однако можно представить четырехугольник в виде двух треугольников (рис. 5.15), а каждый треугольник выводить отдельно. Для каждой треугольной грани перед циклом заполнения нужно определять свои коэффициенты преобразования координат. Сделать это можно по трем точкам вершин треугольника. Рассмотрим треугольник (0-1-2). Очевидно, должны выполняться следующие равенства:

$$x_{T0} = aX_0 + bY_0 + c,$$
$$x_{T1} = aX_1 + bY_1 + c,$$
$$x_{T2} = aX_2 + bY_2 + c,$$

а также

$$y_{T0} = dX_0 + eY_0 + f,$$
$$y_{T1} = dX_1 + eY_1 + f,$$
$$y_{T2} = dX_2 + eY_2 + f.$$

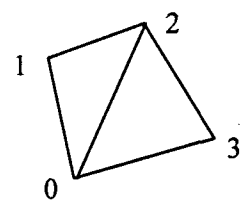


Рис. 5.15. Триангуляция

Рассматривая эти равенства как две системы линейных уравнений третьего порядка относительно (a, b, c) и (d, e, f), можно найти искомые коэффициенты преобразования — подобно тому, как это сделано в разд. 3.7.

Следует заметить, что использовать такой способ триангуляции здесь следует только при большом числе граней, поскольку имеется значительная методическая погрешность. Преимуществом данного способа наложения текстуры является простота вычислительных операций в цикле вывода каждого полигона. В некоторых случаях это позволяет достичь высокой скорости графики.

Хотя теперь полигонов уже почти вдвое больше (грани, примыкающие к полюсам, и так были треугольными), но треугольники обычно выводятся быстрее четырехугольников. Для треугольников часто используют особые алгоритмы, оптимизированные по быстродействию. Кроме того, в цикле вывода грани можно использовать, в основном, целочисленные операции, если соответствующим образом применять коэффициенты аффинного преобразования координат.

Теперь сделаем замечания общего характера по текстурированию.

При наложении текстуры на отдельные грани шара совершенно необязательным является склеивание карты мира в один растр текстуры. Может оказаться удобным для каждой грани иметь отдельную текстуру. Это относится и к случаю, когда текстуры хранятся в файлах.

Для построения реалистичного изображения поверхности Земли в качестве текстуры лучше всего использовать космические снимки, трансформированные надлежащим образом.

Достоинством полигонального текстурирования поверхностей является универсальность метода, применимость его к самым разнообразным поверхностям, которые могут быть представлены многогранниками и полигональными сетями. Еще одним достоинством является высокая скорость графики. Базовые операции текстурирования граней в настоящее время аппаратно поддержаны многими графическими процессорами для видеоадаптеров.

Недостатком метода является погрешность отображения, которая, однако, стремится к нулю при увеличении числа граней.

Вариации формы шара

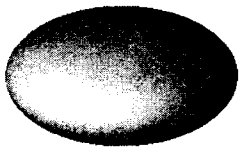
Обозначим через $x_ш$, $y_ш$, $z_ш$ координаты точек поверхности шара. Будем, как и прежде, использовать параметрическую форму — функции угловых координат широты и долготы:

$$x_ш = R \cos B \sin L = F_x(B, L),$$
$$y_ш = R \cos B \cos L = F_y(B, L),$$
$$z_ш = R \sin B = F_z(B).$$

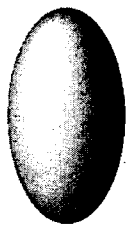
Рассмотрим варианты деформации поверхности шара, которые можно описать следующим образом:

$$x = A_д x_ш + B_д,$$
$$y = C_д y_ш + D_д,$$
$$z = E_д z_ш + F_д,$$

где $A_д$, $B_д$, $C_д$, $D_д$, $E_д$ и $F_д$ — величины, которые могут быть функциями от координат $x_ш$, $y_ш$, $z_ш$, широты и долготы. Рассмотрим некоторые примеры (рис. 5.16—5.22).

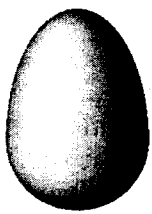


$x = x_ш$,
$y = y_ш$,
$z = 0.5\ z_ш$.



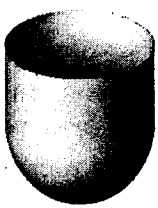
$x = x_ш$,
$y = y_ш$,
$z = 2\ z_ш$.

Рис. 5.16. Эллипсоиды



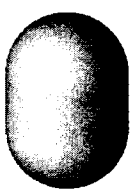
$x = x_ш$,
$y = y_ш$,
$z = 2\ z_ш$, если $z_ш > 0$,
$z_ш$ — иначе.

Рис. 5.17. Наполовину эллипсоид, наполовину — шар



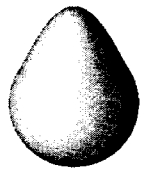
$x = x_ш$,
$y = y_ш$,
$z = R - 0.5\ z_ш$, если $z_ш > 0$,
$z_ш$ — иначе.

Рис. 5.18. Верхняя часть — вогнутый эллипсоид



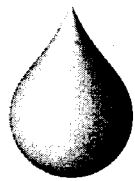
$x = x_ш$,
$y = y_ш$,
$z = z_ш + R$, если $z_ш > 0$,
$z_ш$ — иначе.

Рис. 5.19. Половинки шара разнесены



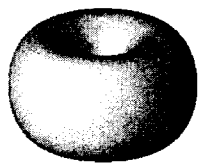
$$x = x_ш,$$
$$y = y_ш,$$
$$z = z_ш + 2.5\,R\,(z_ш/R - 0.5)^2, \text{ если } z_ш > R/2,$$
$$z_ш \text{ — иначе.}$$

Рис. 5.20. "Груша"



$$x = x_ш,$$
$$y = y_ш,$$
$$z = z_ш + R\,(B/90°)^4, \text{ если } B > 0°,$$
$$z_ш \text{ — иначе.}$$

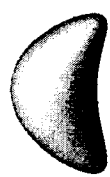
Рис. 5.21. "Капля"



$$x = x_ш,$$
$$y = y_ш,$$
$$z = z_ш - R\,(B/90°)^3.$$

Рис. 5.22. Это не тор

В рассмотренных выше примерах для деформации формы шара были использованы преобразования координат только по оси z. В нижеследующих примерах выполняются преобразования также и координат x, y (рис. 5.23—5.27).



$$x = x_ш + z_ш^2 / R,$$
$$y = y_ш,$$
$$z = 2\,z_ш.$$

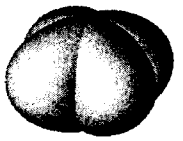
Рис. 5.23. Сдвиг по x пропорционально квадрату z

$$x = x_ш + z_ш^3 / R^2,$$
$$y = y_ш,$$
$$z = 2\,z_ш.$$

Рис. 5.24. Сдвиг по x пропорционально кубу z

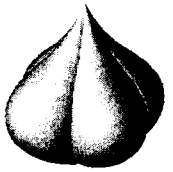
$$x = x_ш + R\,(B/45°)^2,$$
$$y = y_ш,$$
$$z = 2\,z_ш.$$

Рис. 5.25. Сдвиг пропорционален квадрату широты



$$x = x_ш\,(1 + 0.5|\sin(2L)|),$$
$$y = y_ш\,(1 + 0.5|\sin(2L)|),$$
$$z = z_ш.$$

Рис. 5.26. Модуляция *x*, *y* синусом долготы



$$x = x_ш\,(1 + 0.5|\sin(2L)|)\,,$$
$$y = y_ш\,(1 + 0.5|\sin(2L)|)\,,$$
$$z = z_ш + R\,(B/90°)^5, \text{ если } B > 0°,$$
$$z_ш \text{ — иначе.}$$

Рис. 5.27. "Чеснок"

5.2. Цилиндр

Здесь мы будем использовать формулы параметрического описания поверхности цилиндра. В одной из возможных разновидностей такого описания применяются следующие параметры — долгота (*l*) и высота (*h*).

$$x = R \sin\, l,$$
$$y = R \cos\, l,$$
$$z = H\,h,$$

где $l = 0 \ldots 360°$, $h = -0.5 \ldots 0.5$.

Величинами *R* и *H* обозначим соответственно радиус и общую высоту цилиндра (рис. 5.28).

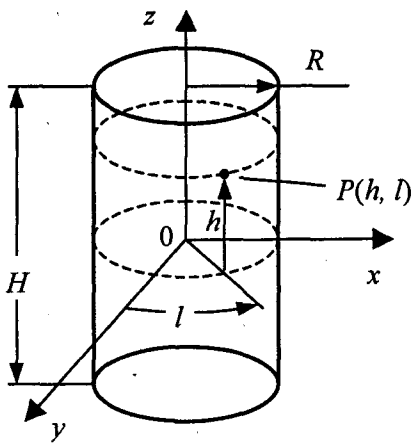
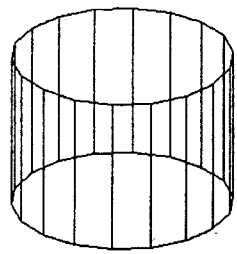


Рис. 5.28. Цилиндр

Каркасное изображение

Построение каркасного изображения полностью аналогично рассмотренному выше способу для шара. Для цилиндра также рисуются меридианы (здесь это вертикальные линии) и параллели (эллипсы). На рис. 5.29 изображен каркас из прямых линий — ребер вписанного многогранника.



Одним из свойств каркасного изображения без удаления невидимых точек является иллюзия неопределенности ракурса. Непонятно — это вид сверху или вид снизу. Раньше вы, наверное, уже замечали подобный эффект, например, когда мы рассматривали каркасное изображение шара.

Рис. 5.29. Каркас

Удаление невидимых точек

На рис. 5.30 показано изображение цилиндра в двух вариантах — с верхней крышкой и без.

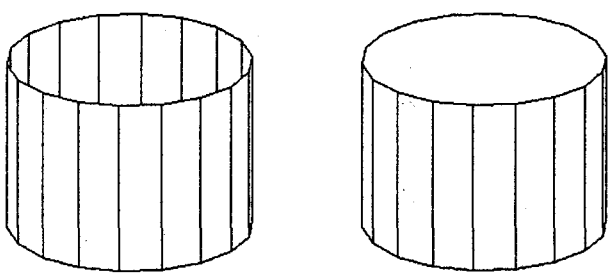


Рис. 5.30. Показ с удалением невидимых точек

В обоих случаях для создания иллюзии объемности здесь выделены черным цветом видимые ребра аппроксимирующего многогранника. Для показа поверхности многогранника с крышкой достаточно показать ребра граней передней стороны и крышку. Задние грани являются невидимыми (рис. 5.31).

Если верхняя крышка отсутствует, то видимыми являются все грани. В этом случае следует рисовать объект, начиная с самых дальних граней задней стороны.

Грани можно выводить попарно, начиная с грани, соответствующей самой дальней точки, как показано на рис. 5.32.

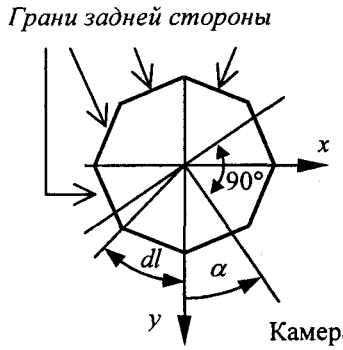


Рис. 5.31. Расположение задних граней. Здесь $dl = 45°$

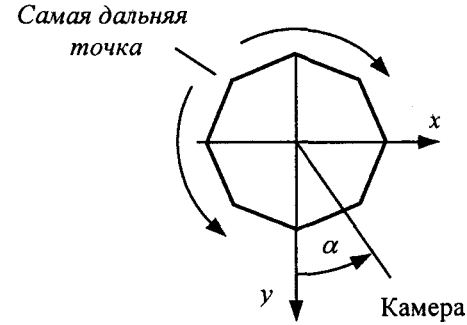


Рис. 5.32. Порядок вывода граней

Долгота дальней точки определяется расположением камеры:

$$l_{st} = dl \times [\,(\alpha - 180°) / dl\,],$$

где [...] — целая часть

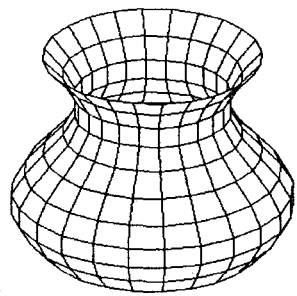


Рис. 5.33. Поверхность вращения

Такой метод сортировки граней можно использовать не только для цилиндра, но и для достаточно широкого класса поверхностей. На рис. 5.33 показан пример поверхности вращения, изображенной описанным выше методом. Такая поверхность подобна цилиндру. Единственное отличие здесь состоит в том, что боковая поверхность криволинейна по вертикали, поэтому для аппроксимации четырехугольными гранями следует использовать двойной цикл – как по долготе (l), так и по высоте (h). Подобные примеры мы рассмотрим несколько позже.

Освещенный многогранник

Здесь мы будем рассматривать изображение боковой поверхности цилиндра, аппроксимированного многогранником. Грани нужно изображать цветом в соответствии с выбранной моделью отражения. Мы будем рассматривать цилиндр без крышек. Это позволяет наглядно проиллюстрировать проблемы показа поверхностей, для которых могут быть видимыми обе стороны.

В чем суть проблемы? Проблема в выборе направления нормали к грани. В предыдущем разделе мы рассматривали шар, и вы, наверное, заметили, что

и там мы аппроксимировали поверхность многогранником. Однако для шара указанной проблемы не существует, у него всегда видимой является внешняя сторона поверхности. Необходимое направление нормалей для всех граней легко обеспечить соответствующей нумерацией вершин каждой грани. Для цилиндра без крышек нельзя жестко зафиксировать направление векторов нормали, если предполагается показывать цилиндр с разных ракурсов (например, изменяя углы поворота камеры α и β).

Для диффузного отражения цвет грани определяется значением косинуса угла между лучом источника света и нормалью к поверхности. Одним из вариантов решения проблемы является взятие модуля косинуса, если косинус отрицательный. Это можно делать, когда положение источника света совпадает с положением камеры. На рис. 5.34 показано несколько вариантов задания направления нормалей.

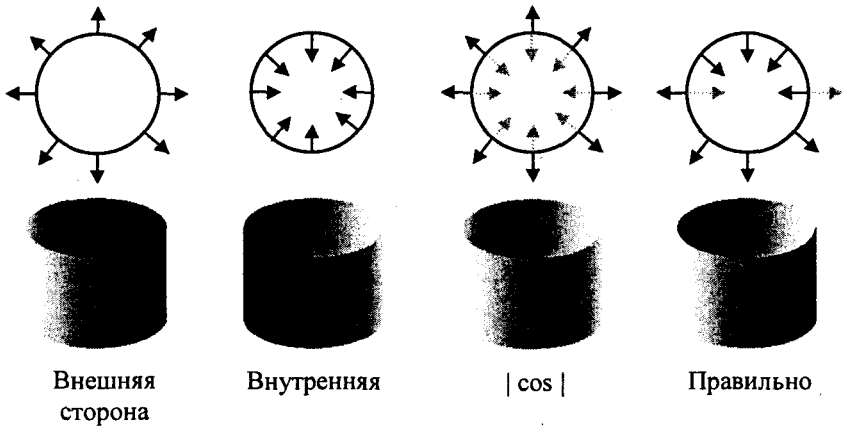


Рис. 5.34. Варианты выбора направления нормалей

На рис. 5.35 изображена боковая поверхность многогранников, аппроксимирующих цилиндр с различной точностью. Моделируется боковое расположение источника света.

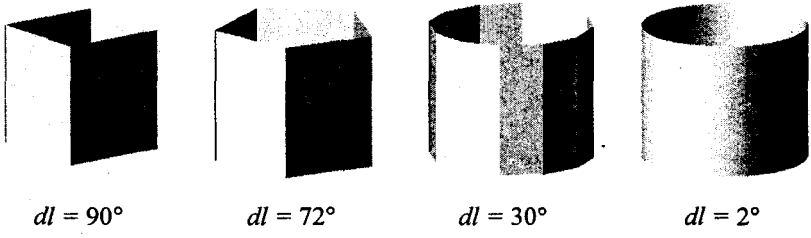


Рис. 5.35. Аппроксимация боковой поверхности плоскими гранями

Гладкая боковая поверхность

Рассмотрим один из алгоритмов изображения гладкой боковой поверхности цилиндра — рисование вертикальными линиями. На рис. 5.36 показаны: вид сверху (ось z мировых координат направлена на нас), полутоновое изображение поверхности в аксонометрической проекции и направление осей мировых координат (x, y, z). Положение камеры здесь задано одним углом β. Источник света располагается так же, как и камера. В этом случае цвет пикселов вертикали боковой поверхности определяется коэффициентом отражения

$$k = \cos\varphi = y / R,$$

где R — радиус цилиндра. Вдоль любой вертикали боковой поверхности цвет пикселов здесь одинаков. Кроме того, при данном расположении источника света имеет место полная симметрия относительно центральной вертикали. Это позволяет построить достаточно быстрый алгоритм рисования. Крышки цилиндра можно также рисовать вертикалями, но это уже не принципиально.

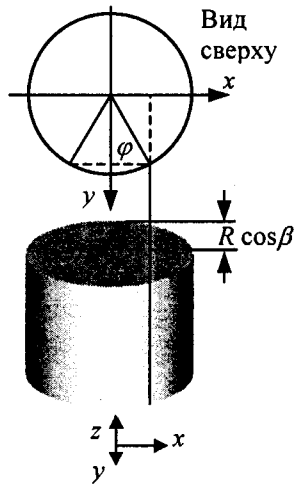


Рис. 5.36. Рисование вертикальными линиями

```
h = H sin β
Определяем цвет крышки (clr0) по значению cos β
for (x = 0; x<=R; x++)
  {
  y = корень квадратный из ( R² - x² )
  k = y / R
  Определяем цвет боковой поверхности (clr) в зависимости от k
  y = y cos β
  Рисование двух симметричных вертикалей боковой поверхности:
     вертикаль от (xc+x, yc+y - h/2) до (xc+x, yc+y + h/2) цветом clr
     вертикаль от (xc-x, yc+y - h/2) до (xc-x, yc+y + h/2) цветом clr
  Рисование двух вертикалей крышки:
     вертикаль от (xc+x, yc-y - h/2) до (xc+x, yc+y - h/2) цветом clr0
     вертикаль от (xc-x, yc-y - h/2) до (xc-x, yc+y - h/2) цветом clr0
  }
```

Здесь (xc, yc) — экранные координаты центра цилиндра.

В ходе работы этого алгоритма вычисляются координаты точек эллипса для одного квадранта. Вначале определяется координата y для круга, а затем она умножается на коэффициент сжатия эллипса ($\cos\beta$). Определение величины $\cos\beta$ целесообразно вынести за цикл — вычислить ее один раз в начале работы. В теле цикла нужно вычислять квадратный корень и выполнять несколько операций умножения и деления. Хотя эти операции являются медленными, но они выполняются сразу для нескольких пикселов, расположенных по симметричным вертикалям. Чем длиннее вертикали, тем меньшую часть от общего времени вывода составляют такие операции (поэтому не обязательно для вычисления точек эллипса использовать алгоритм Брезенхэма — хотя почему бы и нет?). Поскольку собственно вертикаль рисуется достаточно быстро, то данный алгоритм может успешно конкурировать по быстродействию с алгоритмами полигонального вывода. Еще одно полезное свойство — высокое качество рисования гладкой боковой поверхности. Недостатком данного алгоритма является показ только вертикального расположения цилиндра, в то время как полигональную форму можно свободно поворачивать в пространстве.

Следует учитывать, что данный алгоритм можно использовать для углов наклона камеры β от 0 до 90° (вид сверху). Этим и объясняется рисование только верхней крышки. Для вида снизу (β от 90 до 180°) необходимо показывать нижнюю крышку (верхнюю не нужно, она становится невидимой). Алгоритм легко обобщить.

Если цилиндр является отдельным элементом некоторой сложной сцены, то при выводе, например, с использованием Z-буфера необходимо соответствующим образом построить цикл вывода вертикалей.

Теперь рассмотрим произвольное расположение источника света. Как и прежде, будем рассматривать только один точечный источник, расположенный в бесконечности. Пусть расположение источника описывается двумя углами наклона (α_c и β_c). Положение камеры будем задавать двумя углами наклона — (α и β). Для простоты вначале положим угол поворота камеры $\alpha = 0$.

На рис. 5.37 изображен цилиндр, освещенный сбоку. Здесь показана только боковая по-

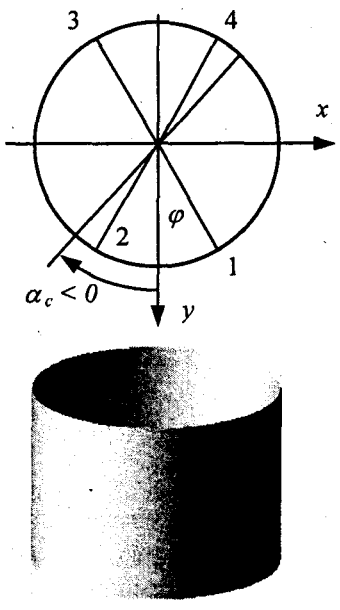


Рис. 5.37. Освещение слева

верхность — без верхней и нижней крышек. Это более сложный вариант показа, поскольку нужно рисовать и внутреннюю часть поверхности.

Рассмотрим четыре точки, расположенные симметрично на поверхности цилиндра. Для первой точки угол между нормалью к боковой поверхности и лучом от источника света составляет $\varphi_1 = \varphi - \alpha_c$. Для второй точки аналогичный угол равен $\varphi_2 = -\varphi - \alpha_c$. Поскольку точки 3 и 4 показываются с внутренней стороны поверхности, то их нормали совпадают по направлению с нормалями соответственно в точках 1 и 2. Поэтому $\varphi_3 = \varphi_1$ и $\varphi_4 = \varphi_2$.

Изображаемая форма цилиндра не зависит от угла поворота камеры α. Этот угол обуславливает лишь закрашивание боковой поверхности (наряду с углами расположения источника света). Учесть угол α можно, если вычесть его из угла поворота источника света. Рассмотрим алгоритм графического вывода.

```
αc = αc - α
h = H sin β
for (x = 0; x<=R; x++)
  {
  y = корень квадратный из ( R² - x² )
  y = y cos β
  φ = arcsin (x / R)
  φ1 = φ - αc
  φ2 = - φ - αc
  Находим коэффициент отражения k1 = cos φ1 sin βc
  если k1 < 0, то k1 = 0
  Находим коэффициент отражения k2 = cos φ2 sin βc
  если k2 < 0, то k2 = 0
  Находим цвета clr1 и clr2, соответствующие k1 и k2
  Рисование двух симметричных вертикалей задней стороны:
     вертикаль от (xc+x, yc-y - h/2) до (xc+x, yc-y + h/2) цветом clr2
     вертикаль от (xc-x, yc-y - h/2) до (xc-x, yc-y + h/2) цветом clr1
  Рисование двух симметричных вертикалей передней стороны:
     вертикаль от (xc+x, yc+y - h/2) до (xc+x, yc+y + h/2) цветом clr1
     вертикаль от (xc-x, yc+y - h/2) до (xc-x, yc+y + h/2) цветом clr2
}
```

Здесь для упрощения записи алгоритма использована идеальная диффузная модель отражения света. Вначале определяются коэффициенты k_1 и k_2, которые затем можно использовать для вычисления цвета, например, в виде компонент R, G, B:

$$\mathbf{clr1} = (\mathbf{k}_1 R_ц,\ \mathbf{k}_1 G_ц,\ \mathbf{k}_1 B_ц),$$
$$\mathbf{clr2} = (\mathbf{k}_2 R_ц,\ \mathbf{k}_2 G_ц,\ \mathbf{k}_2 B_ц),$$

где $R_ц$, $G_ц$ и $B_ц$ — компоненты описания цвета поверхности цилиндра.

Вертикали выводятся сначала для задней стороны боковой поверхности, а затем для передней. При использовании Z-буфера порядок вывода безразличен.

Данный алгоритм сложнее предыдущего. Здесь в теле цикла выполняется значительное количество медленных операций — кроме корня квадратного вычисляется также и арксинус. Это является недостатком алгоритма с позиций быстродействия. Положительной чертой алгоритма является вычисление угловой координаты (φ), которую можно использовать для наложения текстуры-развертки (карты).

Наложение текстуры

В некоторой степени, компьютерная графика — это создание иллюзий. Текстуры наглядно это демонстрируют (рис. 5.38). Разве бывают кирпичи округлой формы? Крайний справа цилиндр "выложен" именно такими кирпичами. При наложении текстуры плоская карта-развертка растрового образца текстуры плавно искривляется, следуя гладкой форме боковой поверхности.

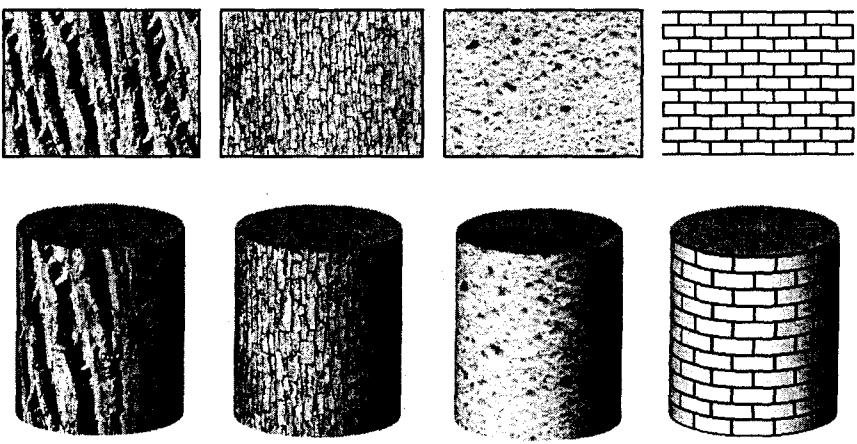


Рис. 5.38. Наложение текстур на боковую поверхность цилиндра

Попробуйте подобрать текстуру, чтобы кирпичи выглядели действительно плоскими, как в реальной жизни. Однако тогда может оказаться заметным искривление в другом ракурсе. Текстура — это достаточно простой способ

изображения сложных поверхностей, это начальный уровень имитации материалов. В самом деле, если строить модель каждого отдельного кирпича или углублений шероховатой поверхности, то эта модель будет намного сложнее цилиндра. Таким образом, для создания более или менее правдоподобного изображения, которое в отдельных случаях удается выдавать в качестве реалистичного, необходимо, в первую очередь, подобрать соответствующую текстуру.

Кроме того, и для текстурированных объектов очень важно соблюдать законы отражения. Если убрать градиентную закраску у четырех цилиндров, изображенных на рис. 5.38, то для первых трех (кроме "кирпичного") почти полностью исчезнет иллюзия объемности.

Поскольку текстура чаще всего представляет собой оцифрованный фотоснимок реальных объектов, сделанный с определенного ракурса съемки, то текстурирование дает удовлетворительный результат только для соответствующих ракурсов показа.

Рассмотрим, как наложить текстуру на боковую поверхность цилиндра. Способ наложения определяется типом алгоритма рисования. Если цилиндр рисуется как многогранник, то в этом случае текстурирование выполняется так, как нами уже было рассмотрено выше для многогранников, аппроксимирующих шар. Алгоритмы рисования объектов в виде многогранников и полигональных сетей являются достаточно универсальными и могут быть использованы для широкого класса объектов.

Здесь мы рассмотрим один способ наложения текстуры, специально предназначенный для цилиндра. В его основе лежит способ рисования цилиндра вертикалями, рассмотренный выше. На рис. 5.39 показана одна из вертикалей закрашивания поверхности.

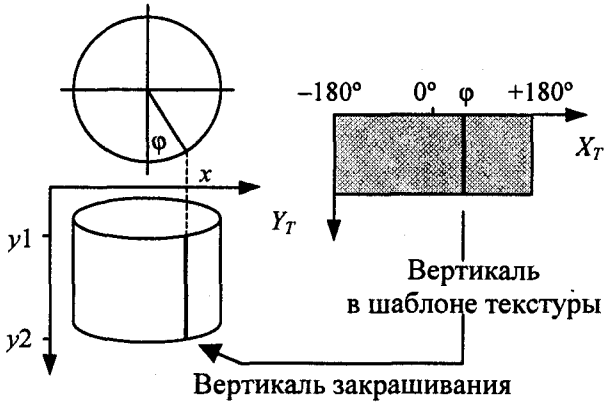


Рис. 5.39. Наложение текстуры вертикалями

Общий цикл рисования боковой поверхности можно представить себе следующим образом. Вначале определяются координаты текущей вертикали закрашивания в экранной системе координат. Пусть эти координаты составляют (x, $y1$) – (x, $y2$). Необходимо также вычислить угловую координату этой вертикали (φ). По значению угла φ определяем координату X_T в шаблоне текстуры

$$X_T = (\varphi + 180)\frac{HorTex - 1}{360},$$

где *HorTex* — горизонтальный размер шаблона текстуры. Угол φ измеряется в градусах.

Одна вертикаль закрашивания (x, $y1$) – (x, $y2$) соответствует вертикали с координатами (X_T, 0) – (X_T, *VertTex* – 1) в шаблоне текстуры, где *VertTex* — размер текстуры по вертикали.

Рисование одной вертикали боковой поверхности можно представить как цикл, в ходе которого последовательно рисуются пикселы с координатами y от $y1$ до $y2$. Для пиксела в точке (x, y) определяются соответствующие координаты (X_T, Y_T) в шаблоне текстуры, причем X_T нам уже известна, а координата Y_T вычисляется по формуле:

$$Y_T = (y - y1)\frac{VertTex - 1}{y2 - y1}.$$

Затем в шаблоне текстуры определяется цвет точки (X_T, Y_T). После этого выбранным цветом рисуется пиксел с координатами (x, y) в основном растре.

Формулы вычисления координат можно преобразовать к следующему виду:

$$X_T = A\varphi + B,$$
$$Y_T = Cy + D,$$

где A, B, C и D — константы для всех пикселов одной вертикали. Таким образом, для каждого пиксела вертикали приходится выполнять несколько операций умножения и сложения. Для ускорения текстурирования можно построить инкрементный алгоритм.

Данный способ наложения текстуры можно использовать только для простейших цилиндров, расположенных вертикально. При наложении текстур на более сложные поверхности (в том числе вариации формы цилиндра) следует применять полигональные методы.

Вариации формы цилиндра

Параметрические формулы цилиндра удобно использовать в качестве основы для описания поверхностей достаточно сложной формы. В исходных параметрических уравнениях цилиндра

$$x = R \sin l,$$
$$y = R \cos l,$$
$$z = H h,$$

величины H и R — это константы. Рассмотрим примеры поверхностей, когда радиус R является функцией параметров h и l, то есть $R = R(h, l)$.

Если радиус зависит только от высоты, то есть $R = R(h)$, то это соответствует поверхности вращения относительно оси z. Задание конкретной функции $R(h)$ для описания какой-либо поверхности напоминает вытачивание цилиндрической заготовки на токарном станке. На рис. 5.40 приведены примеры поверхностей вращения (во всех случаях параметр h изменяется от –0.5 до +0.5).

$$R = R_1 + (R_2 - R_1)(h + 0.5),$$

где R_1 и R_2 — радиусы нижней и верхней части соответственно. При $R_2 = 0$ получаем конус. Очевидно, что при аппроксимации многогранниками конус следует рисовать уже не четырехугольными, а треугольными гранями.

$$R = R_1 + 2(R_2 - R_1) \mid h \mid.$$

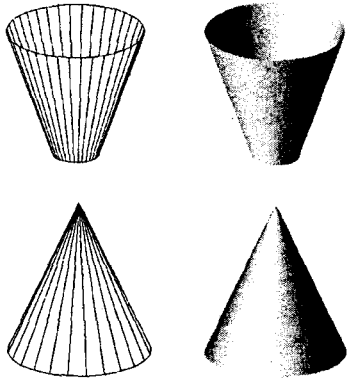


Рис. 5.40. Примеры поверхностей вращения

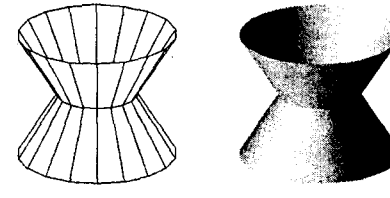


Рис. 5.41. Еще пример поверхности вращения

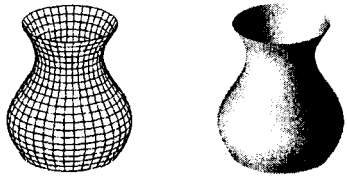
Для поверхности, изображенной на рис. 5.41, при вычислении координат вершин четырехугольных граней необходимо использовать уже два цикла — по l и по h.

Например:

```
for (h = -0.5;   h < 0.5;   h = h+dh)
   for (l = 0;   l<360°;   l = l+dl)
      {
      P0 = Точка Поверхности (h, l)
      P1 = Точка Поверхности (h+dh, l)
      P2 = Точка Поверхности (h+dh, l+dl)
      P3 = Точка Поверхности (h, l+dl)
      Вывод грани ( P0, P1, P2, P3)
      }
```

В данном случае вдоль вертикали располагаются по две грани ($dh = 0.5$).

В следующих примерах для создания иллюзии гладкой поверхности нужно использовать большее число граней (рис. 5.42, 5.43).



$R = R_c\,(1 - 0.3 \sin(2h\pi))$,
где R_c — константа.

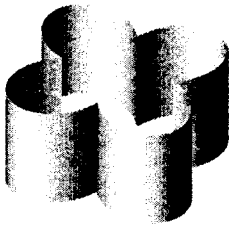
Рис. 5.42. Сетка граней и закрашивание методом Гуро

$R = R_c\,(1 + |\sin(2h\pi)|)$ $\qquad$ $R = R_c\,(1 - 0.3|\sin(4h\pi)|)$

Рис. 5.43. Еще две поверхности вращения

Следующую группу составляют такие вариации формы цилиндра, когда радиус зависит только от долготы, то есть $R = R(l)$. Пример подобной поверхности показан на рис. 5.44.

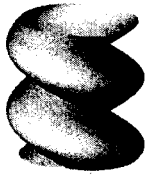
И, наконец, последнюю разновидность вариаций данного типа, согласно нашей классификации, представляют поверхности $R = R(h, l)$. Пример такой поверхности приведен на рис. 5.45.



$$R = R_c\,(1 + |\sin(2l)|),$$

где l — долгота от 0 до 360 градусов.

Рис. 5.44. Другая разновидность вариаций формы цилиндра



$$R = R_c\,(1 + |\sin(2h\pi + 0.5\,l)|)$$

Рис. 5.45. Винтовая поверхность $R = R(h, l)$

5.3. Тор

Функции параметрического описания поверхности тора запишем в следующем виде

$$x = (R + r\cos\varphi)\sin\omega,$$
$$y = (R + r\cos\varphi)\cos\omega,$$
$$z = r\sin\varphi,$$

где R и r — большой и малый радиусы, φ и ω — широта и долгота. Для замкнутой поверхности углы φ и ω должны изменяться в полном круговом диапазоне, например, от 0 до 360° или от –180° до +180°.

На рис. 5.46, 5.47 показаны различные способы изображения тора.

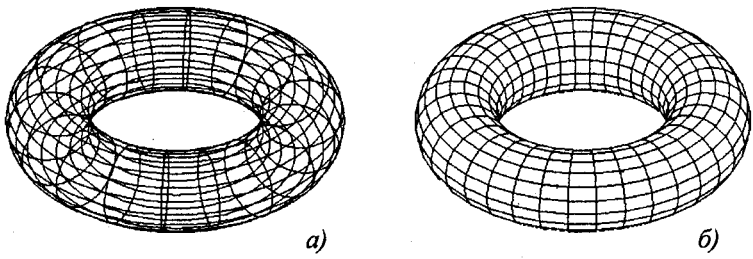


Рис. 5.46. Простейшее изображение тора:
а — каркас; *б* — поверхность с удаленными невидимыми точками

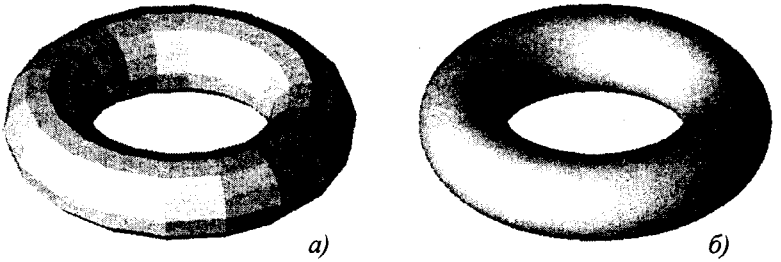


Рис. 5.47. Многогранники, диффузная модель отражения:
а — обычное закрашивание; *б* — интерполяция по методу Гуро

Рисование тора средствами компьютерной графики достаточно просто может быть выполнено на основе аппроксимации многогранником, подобно тому, что мы уже рассматривали для шара и цилиндра. Повторим запись алгоритма вывода многогранника четырехугольными гранями, видоизменив его для данного конкретного случая

```
for  (ω = -180;  ω < 180; ω=ω+dω)
   for  (φ = -180;  φ < 180; φ=φ+dφ)
     {
       Вычисление координат точки P0 для (φ, ω);
       Вычисление координат точки P1 для (φ+dφ, ω);
       Вычисление координат точки P2 для (φ+dφ, ω+dω);
       Вычисление координат точки P3 для (φ, ω+dω);
       Определение необходимых атрибутов закрашивания.
       Рисование четырехугольника  (P0 , P1 , P2 , P3 ).
     }
```

Очевидно, что чем меньше величины $d\omega$ и $d\varphi$, тем больше число граней у вписанного многогранника и тем лучше такой многогранник соответствует гладкой поверхности тора. При использовании такого полигонального метода получения изображения достаточно просто наложить текстуру. Полигональный способ наложения текстур для тора полностью аналогичен способу, рассмотренному выше для шара. Пример подобного текстурирования приведен на рис. 5.48.

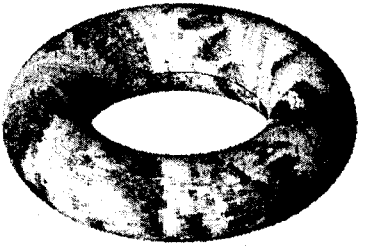


Рис. 5.48. Тор с текстурой

Рассматривая в предыдущих разделах шар и цилиндр, мы наряду с полигональными способами пытались анализировать и другие способы изображе-

ния. Например, рисование цилиндра вертикальными линиями. Для тора изобрести эффективный специальный алгоритм графического вывода, вероятно, достаточно сложно. Во всяком случае, автору он неизвестен. Хотя во второй части этой книги есть пример программирования одного "неполигонального" способа рисования тора. Можно представить тор как след движения шара. Если шар перемещается с достаточно малым шагом, то след получается весьма похожим на тор. Впрочем, данный пример рисования приведен в книге не как рекомендуемый способ изображения тора, а как пример изображения движущихся шариков. По быстродействию данный способ изображения тора весьма плох из-за того, что приходится многократно рисовать одни и те же точки (подобные аспекты мы рассматривали в главе 3 при обсуждении алгоритмов рисования толстых линий).

Вариации формы тора

На рис. 5.49 изображена поверхность многогранника, для которой параметрические формулы такие же, как и для тора. Единственное отличие здесь в том, что широта φ изменяется в диапазоне от –135° до +225° с шагом $d\varphi = 90°$.

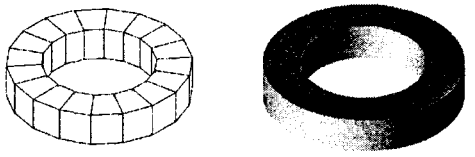


Рис. 5.49. Кольцо

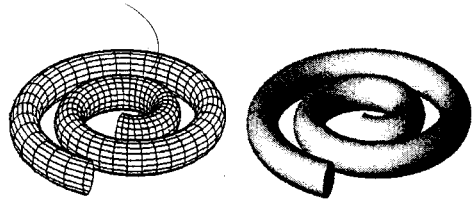


Рис. 5.50. Спираль

Если изменять радиус R пропорционально долготе, т. е. $R = R(\omega)$, то получим спираль (рис. 5.50). Здесь больший диапазон изменения долготы: от –360° до +360°, соответствует двум виткам спирали.

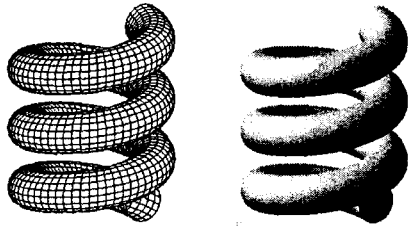


Рис. 5.51. Пружина

Для пружины (рис. 5.51) значения координат x и y такие же, как и для тора, а координата z тора суммируется с приращением, пропорциональным долготе:

$$x = (R + r\cos\varphi)\sin\omega,$$

$$y = (R + r\cos\varphi)\cos\omega,$$

$$z = r\sin\varphi + k\,\omega,$$

где k — некоторая константа, определяющая шаг витков спирали по высоте. Долгота здесь изменяется в троекратном круговом диапазоне (соответствующем числу витков).

Если объединить спираль и пружину, то получим коническую спираль (рис. 5.52):

$$x = (R + p\omega + r\cos\varphi)\sin\omega,$$
$$y = (R + p\omega + r\cos\varphi)\cos\omega,$$
$$z = r\sin\varphi + k\omega,$$

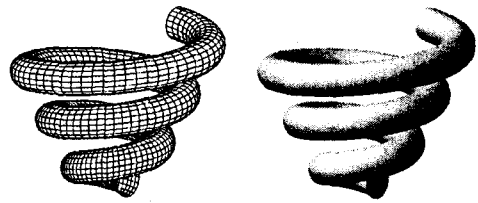


Рис. 5.52. Коническая спираль

где константа p определяет увеличение большого радиуса пропорционально долготе, а k — задает шаг витков пружины по высоте.

5.4. Общие замечания

Мы рассмотрели шар, цилиндр и тор, а также их некоторые вариации. Очевидно, что это лишь малая часть всего многообразия трехмерных форм. Тем не менее, можно сделать некоторые обобщения.

Модель описания и способ отображения

Для описания формы поверхности мы использовали аналитическую модель — параметрические формулы

$$x = F_x(s, t),$$
$$y = Fy(s, t),$$
$$z = Fz(s, t),$$

где s и t — параметры, непрерывно изменяющиеся в некотором диапазоне.

В качестве одного из возможных способов построения изображения поверхности мы рассматривали такой способ. В процессе рисования выполняется цикл

```
for (s = smin;  s < smax; s = s+ds)
    for (t = tmin;  t < tmax;  t = t+dt)
       {
       P0 = точка поверхности (s, t) ;
       P1 = точка поверхности (s+ds, t) ;
```

```
P2 = точка поверхности (s+ds, t+dt) ;
P3 = точка поверхности (s, t+dt) ;
Определение атрибутов закрашивания грани
{
Вычисление нормали к грани (P0,P1,P2,P3) или же при интерполяции
методом Гуро или Фонга вычисление нормали в вершинах грани
(аналитически — вычислением частных производных в соответствующих
точках поверхности либо усреднением нормалей к соседним граням).
Определение цветовых атрибутов закрашивания грани.
}
Вычисление экранных координат вершин P0,P1,P2 и P3
Вывод полигона (P0,P1,P2,P3)
}
```

Таким образом, здесь вычисляются координаты узлов сетки с шагом *ds* и *dt*. Поверхность изображается в виде четырехугольных граней. Чем меньше шаг сетки, тем больше таких граней и тем лучше соответствие форме гладкой поверхности.

Является ли данный способ отображения поверхности наилучшим? Вряд ли. Здесь каждая точка поверхности вычисляется четырежды, что существенно замедляет процесс, если поверхность описана сложными формулами. Является ли данный способ самым простым? Вероятно. Можно также подчеркнуть его экономичность по затратам памяти, поскольку не нужны массивы для хранения координат граней — координаты вычисляются "на ходу". Но главное, что следует подчеркнуть, — этот способ универсален. Его можно применить для отображения широкого класса поверхностей. Он может использоваться для произвольных поверхностей, для которых известны параметрические формулы, например для сплайнов. На рис. 5.53 и 5.54 показаны несколько вариантов изображения кубического сплайна Безье рассматриваемым способом.

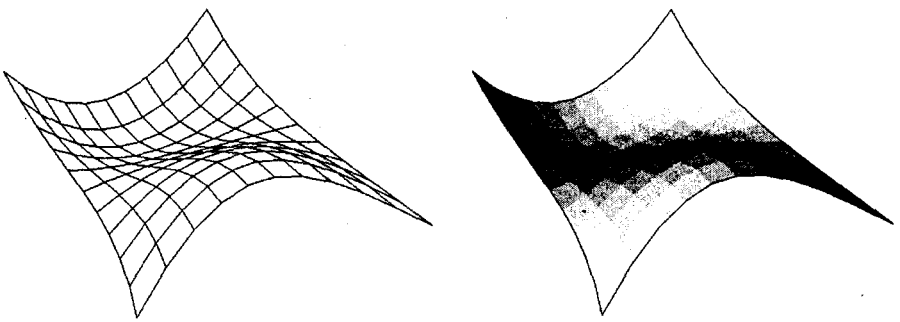


Рис. 5.53. Аппроксимация гладкой поверхности сплайна

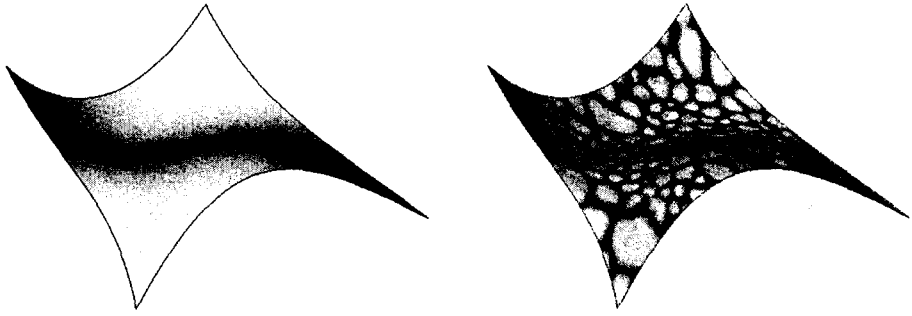


Рис. 5.54. Закраска Гуро и наложение текстуры

В компьютерной графике важную роль играет быстродействие вывода. Повысить скорость рисования для рассматриваемого базового алгоритма можно следующим образом. Чтобы избежать повторного вычисления координат вершин граней, вначале следует вычислить узловые координаты всех точек поверхности и записать в массив. Затем можно в отдельный массив записать координаты векторов нормалей в вершинах. Потом преобразовать координаты и рисовать полигоны граней. Введение дополнительных массивов для вершин и нормалей обычно заметно увеличивает скорость, например, при закрашивании Гуро, поскольку исключаются повторные расчеты не только координат вершин, но и векторов нормалей к соседним граням.

Однако общий характер цикла отображения здесь сохраняется. Когда нужно отобразить требуемый объект, то заново вычисляются все узловые координаты. А теперь представим себе, что некоторая трехмерная сцена содержит несколько таких объектов. Предположим, что многие объекты остаются неподвижными, а изменяется лишь ракурс показа. Зачем тогда всякий раз вычислять мировые (а точнее, некоторые локальные) координаты по параметрическим формулам? Их можно вычислить только один раз при инициализации сцены и записать координаты всех вершин (и нормалей, если необходимо) в массивы. А при расчете кадров производить только все необходимые преобразования координат и закрашивание. Более того, можно параметрические формулы не вычислять вообще, если хранить описание координат вершин (и нормалей) в файлах соответствующего формата, которые загружаются при инициализации сцены. Так мы постепенно перешли от аналитической модели описания объектов к полигональной модели, которая также имеет свои плюсы и минусы.

Модель описания и способ отображения, вообще говоря, не обязательно соответствуют друг другу.

"Квадратирование" и триангуляция

В качестве небольшого околонаучного развлечения давайте попрактикуемся в терминологии. Рассматриваемый здесь способ отображения поверхностей четырехугольными гранями можно назвать "квадратированием". Способ этот известен давно, а такое название мы с вами можем изобрести и сами. Оно кажется достаточно звучным и вполне соответствующим сути, а главное — может помочь застолбить "непоправимый вклад в науку" (последнее выражение в кавычках придумано не мною). А если говорить серьезно, то лучше все-таки следовать общему правилу — не засорять язык без крайней необходимости. Поэтому используем кавычки.

Применение четырехугольных граней для поверхностей общего вида, вообще говоря, это вопрос спорный. Почему именно четырехугольные грани? Для сравнения на рис. 5.55 и 5.56 приведены два варианта граней — четырехугольники и треугольники.

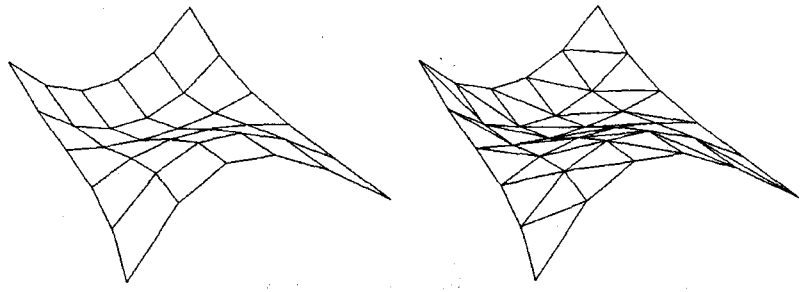


Рис. 5.55. "Квадратирование" и триангуляция

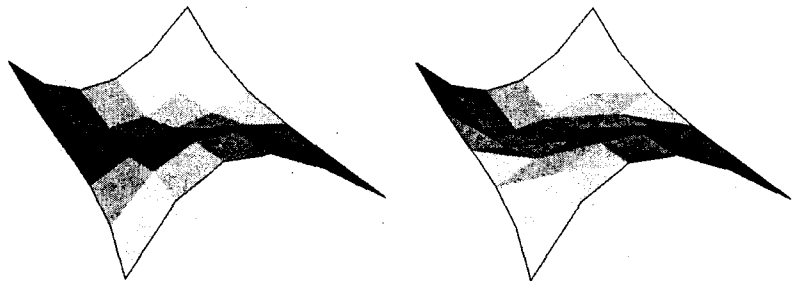


Рис. 5.56. При небольшом числе граней видны существенные отличия

Здесь необходимо упомянуть следующие понятия. Грань называется *плоской*, если все ее вершины располагаются в одной плоскости трехмерного пространства. При отображении трехмерного объекта на плоскости пространст-

венные грани (как плоские, так и неплоские) изображаются закрашенными многоугольниками — полигонами. Полигоны бывают выпуклыми и невыпуклыми. *Выпуклый* полигон — это фигура на плоскости, контур которой пересекается любой прямой линией только дважды. Если находится такая прямая линия, которая пересекает контур большее число раз, то это невыпуклый полигон. Это обуславливает особенности алгоритмов графического вывода. Алгоритмы вывода полигонов мы рассматривали в главе 3.

Очевидно, что любой треугольник всегда выпуклый. Таким образом, треугольная грань в пространстве всегда является плоской и отображается на плоскости выпуклым полигоном. Четырехугольная грань может быть как плоской, так и неплоской. И та и другая четырехугольная грань может отображаться в проекции как выпуклым, так и невыпуклым полигоном. Любой четырехугольник можно изобразить в виде двух треугольников. Некоторые примеры таких полигонов приведены на рис. 5.57.

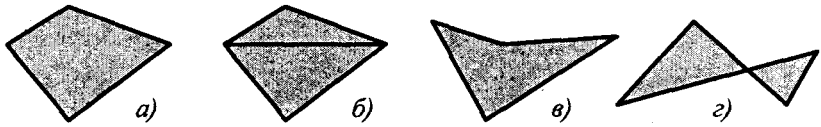


Рис. 5.57. Изображение четырехугольных граней:
а — выпуклый полигон, *б* — триангуляция, *в* и *г* — невыпуклые полигоны

Для поверхностей вращения все четырехугольные грани являются плоскими (разумеется, если поверхность аппроксимировать так, как это сделано в предыдущих разделах — по меридианам и параллелям). Если форма — не поверхность вращения, то такие грани могут быть плоскими, а могут таковыми и не быть. Это можно отметить и для некоторых вариаций формы шара, цилиндра и тора, приведенных выше. Зачем нужны именно плоские грани? И когда необходимо применять триангуляцию? Для ответа на эти вопросы нужно учитывать многие аспекты.

Алгоритм отображения может вносить некоторую погрешность — в одних случаях заметную, а в некоторых случаях ею можно пренебречь. Триангуляция в этом плане является более корректной. В то же время, иногда изображение, построенное из треугольных граней, субъективно выглядит существенно хуже, чем при использовании четырехугольников.

С другой стороны, неплоские грани могут значительно затруднить процесс формирования изображения из-за ограничений используемой технологии. В разных графических системах в этом плане имеются существенные отличия. Так, например, при определенном ракурсе показа неплоская четырех-

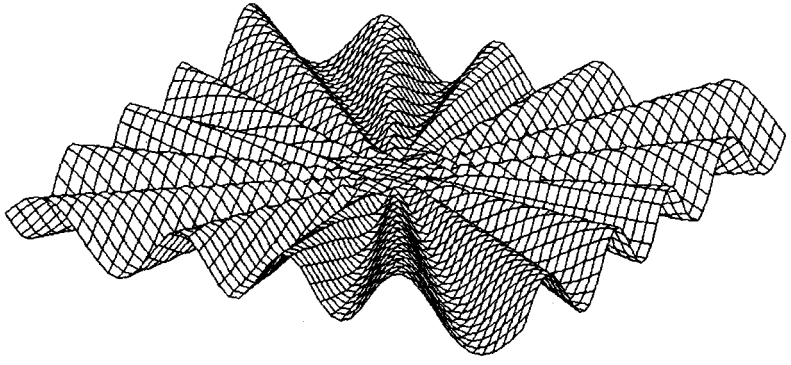
угольная грань изображается невыпуклым полигоном. Это и приводит к проблемам. Например, в API библиотеки OpenGL в качестве графических примитивов предлагаются только выпуклые полигоны, а в API Windows полигон-примитив может быть и невыпуклым. Очевидно, что любой невыпуклый полигон может быть разрезан на треугольники (или другие выпуклые фигуры), но это усложняет алгоритм отображения.

Существенную роль играет быстродействие. Например, если в некоторой графической системе два треугольника выводятся быстрее, чем один четырехугольник, то при прочих равных условиях это может оказаться решающим фактором.

При аппроксимации поверхностей не обязательно использовать постоянный шаг сетки. Размеры и ориентация граней могут варьироваться в зависимости от кривизны участков поверхностей. Это позволяет достичь лучшего соотношения между точностью аппроксимации и количеством граней.

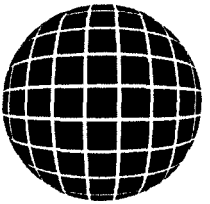
ЧАСТЬ II

Программирование компьютерной графики



Глава 6.	Разработка графических программ для Windows
Глава 7.	Графические примитивы API Windows
Глава 8.	Примеры использования классов языка C++
Глава 9.	Пример анимации
Глава 10.	Графическая библиотека OpenGL

ГЛАВА 6



Разработка графических программ для Windows

Для разработки разнообразных программ для операционной системы Windows существует много инструментальных средств. Различные средства могут воплощать в практику различные методологические подходы. Одну и ту же программу можно изготовить, как правило, несколькими способами. В том числе и графическую программу.

Мы будем использовать язык программирования C++, а если говорить точнее, то преимущественно обычный C. Везде, где это возможно, будем обходиться простыми языковыми средствами. Такие элементы C++, как классы, будем использовать только там, где они действительно необходимы, и где без них трудно обойтись. Не следует считать это отказом от современных технологий, например, от объектно-ориентированного и компонентного программирования, визуальной технологии разработки программ. Я с большим уважением отношусь к этим действительно эффективным технологиям, но здесь они практически бесполезны (во всяком случае, последние две).

Для того чтобы сосредоточиться на особенностях программирования именно графики, при написании наших учебных программ мы не будем отвлекаться на другие аспекты программирования (например, разработку пользовательского интерфейса). Такое упрощение, как представляется, позволит детальнее ознакомиться с компьютерной графикой изнутри. Если не знать графику, то компоненты и визуальное программирование не помогут. Последнее утверждение может показаться явно спорным, однако, по-моему, это именно так — не следует писать программы, не понимая сути того, что программируешь.

Графическая программа, как конечный продукт, с одной стороны, воплощает некоторые обобщенные методы и алгоритмы. С другой стороны, эти методы и алгоритмы должны обязательно соответствовать архитектуре компьютера и

операционной системы. Разумеется, можно писать программы, не зная, как устроен процессор и как работает память, однако это непрофессионально.

Важный аспект при разработке программ — это изучение особенностей операционной системы (ОС) — в нашем случае это будет Windows. Известно несколько библиотек, например MFC или OWL, которые прячут детали функционирования Windows, и может показаться, что успешно программировать можно без изучения операционной системы. Вместо этого изучается версия библиотеки, считающаяся на данный момент современной. Но библиотеки изменяются, некоторые из них вообще исчезают, и тогда необходимо изучать очередную библиотеку и инструментальную среду разработки. Так рано или поздно приходит осознание того, что изучение самой операционной системы (которая существует значительно дольше, чем любая из библиотек и сред программирования для этой ОС) и архитектуры компьютера закладывает надежный фундамент для дальнейшего повышения квалификации, в том числе и для овладения новыми технологиями программирования.

Мы будем изучать использование функций API Windows без посредничества высокоуровневых библиотек общего назначения для разработки программ. Потом рассмотрим графическую библиотеку OpenGL.

Из множества литературных источников, освещающих эту тему, можно порекомендовать в первую очередь документацию SDK для разработчиков Windows-программ [61] и прекрасную книгу [20].

6.1. Первый пример программы для Windows

Наша первая графическая программа названа StudEx. Она предназначена для работы в среде 32-битных ОС Windows различных версий, например, 95, 98, 2000, Millennium, NT. Программа написана на языке C++ (строго говоря, почти что на чистом C, если не считать комментариев) с использованием функций API Windows. Текст программы состоит из трех файлов: `studex.cpp, studex.rc, studex.def.`

Файл `studex.cpp` (это главный файл текста программы):

```
//----------учебный пример программы для Windows-----------
#define STRICT
#include <windows.h>
//-------------Объявление функций (прототипы)--------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam);
void DrawStudyExample(HWND hWnd);
```

```
//---------Затем следует определение функций---------------
//---------------Главная функция программы------------------
//-----Она определяет первые шаги работы программы----------
//----до создания окна и вхождения в цикл сообщений---------
//--для Windows-программ функция C,C++ main названа WinMain
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCmd,
                    int nCmdShow)
{
MSG msg;
HWND hWnd;
WNDCLASS WndClass;

//---сначала регистрируем класс главного окна программы--
WndClass.style         = NULL;
WndClass.lpfnWndProc   = WndProc;    //адрес функции окна
WndClass.cbClsExtra    = 0;
WndClass.cbWndExtra    = 0;
WndClass.hInstance     = hInstance;
WndClass.hIcon         = LoadIcon( NULL, IDI_APPLICATION );
WndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
WndClass.hbrBackground =(HBRUSH)GetStockObject(WHITE_BRUSH);
WndClass.lpszMenuName  = "STUDEXMENU";
WndClass.lpszClassName = "StudEx";
if (!RegisterClass(&WndClass)) return 0;

//------------потом создаем окно класса StudEx----------
hWnd = CreateWindow("StudEx",
                    "Учебный пример",       //заголовок
                    WS_OVERLAPPEDWINDOW, //стиль окна
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    400,                 //размеры окна
                    300,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);
if (!hWnd) return NULL;            //окно создано успешно?
ShowWindow(hWnd, nCmdShow);             //отобразить окно
UpdateWindow(hWnd);                     //обновить окно
```

```
//------------организация цикла обработки сообщений-------
while (GetMessage(&msg,NULL,NULL,NULL))
  {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
  }
return msg. wParam;
}

//--------------функция главного окна программы----------
//--ее вызывает Windows, когда посылает сообщение нашей--
//-------------------------программе--------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)   //Обработка всех сообщений для окна
  {
  case WM_COMMAND:       //Обработка сообщений пунктов меню
    switch (LOWORD(wParam))
      {
      case 201:              //выбран пункт меню Графика
        DrawStudyExample(hWnd); //это наша собственная
                                //функция
        break;
      case 108:
        DestroyWindow(hWnd);    //для завершения работы
        break;
      default : break;
      }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);  //для окончания цикла сообщений
    break;
  default : return DefWindowProc(hWnd,       //обработка
                                 message,    //всех
                                 wParam,     //остальных
                                 lParam);    //сообщений
  }
return 0L;
}

//------------это наша собственная функция, она ------------
//-будет содержать графические операторы рисования в окне---
void DrawStudyExample(HWND hWnd)
```

```
{
HDC hdc;

hdc = GetDC(hWnd);                    //контекст окна
if (hdc)
   {
   //после того, как получено ненулевое hdc, можно рисовать,
   //например, десять прямоугольников размером 70 x 50 ...
   for (int i=0; i<10; i++)
       Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
   }
ReleaseDC(hWnd,hdc);     //этот контекст больше не нужен
}
```

Файл `studex.rc` (здесь содержится описание меню):

```
STUDEXMENU MENU
{
POPUP "&Файл"
  {
  MENUITEM "&Открыть...", 101
  MENUITEM "&Записать как...", 102
  MENUITEM SEPARATOR
  MENUITEM "&Печать", 103
  MENUITEM SEPARATOR
  MENUITEM "&Закончить", 108
  }
MENUITEM "&Графика",  201
}
```

Файл `studex.def` (файл описания для Windows):

```
NAME StudEx
DESCRIPTION 'Study Example'
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
```

Приведенный выше текст программы — это только исходный текст. Его необходимо скомпилировать, чтобы получить выполняемый файл в машинных кодах. Для компиляции и отладки этой программы можно использовать разнообразные инструментальные средства программирования, например, Borland C++ 5.02, Borland C++ Builder, Microsoft Visual C++ 6.0 и другие. После создания проекта в среде системы программирования в результате компили-

рования получим выполняемый файл. Имя этого файла — `studex.exe` (или другое, что определяется при создании проекта). Запустите программу в среде Windows. После появления окна программы выберите пункт меню "Графика". На экране дисплея будет изображение, как на рис. 6.1.

Рис. 6.1. Первый пример графической программы

6.2. Модульность программ

Приведенный выше текст первого примера графической программы для ОС Windows достаточно объемный. Для того чтобы упростить тексты других примеров графических программ, необходимо спланировать наше дальнейшее программирование. Рассмотрим структуру главного файла (`studex.cpp`):

```
.........              // начало текста, объявление функций
WinMain                // главная функция
```

```
WndProc                // оконная функция
..........
```

```
DrawStudyExample       // другие функции (пока что одна)
..........
```

Как видим, программа состоит из трех частей. Первая из них, содержащая функцию `WinMain`, будет присутствовать во всех следующих примерах без всяких изменений. Это достаточно длинный текст, поэтому вместо того, чтобы его каждый раз копировать во всех следующих примерах, создадим отдельный файл, на который будем ссылаться в дальнейшем, и назовем его `winmain.cpp`:

```
//-------------функция WinMain и объявление WndProc ------
#define STRICT
#include <windows.h>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam);
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCmd,
                    int nCmdShow)
{
. . . .         //тело функции WinMain
}
```

В следующих примерах мы будем использовать различные варианты оконной функции `WndProc`. Тот вариант, что приведен для первого примера, будет часто использоваться — его также запишем в отдельный файл и назовем `winmain1.cpp`:

```
#include "winmain.cpp"
void DrawStudyExample(HWND hWnd);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
  {
  case WM_COMMAND:
    switch (LOWORD(wParam))
      {
      case 201:
        DrawStudyExample(hWnd);
        break;
      case 108:
        DestroyWindow(hWnd);
        break;
      default : break;
      }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
   default:return DefWindowProc(hWnd,message,wParam,lParam);
  }
return 0L;
}
```

В файле `winmain1.cpp` содержится текст оконной функции, объявление функции `DrawStudyExample` и директива `#include "winmain.cpp"`, благодаря которой при компиляции используется текст функции `WinMain`. Следовательно, текст первого примера программы можно трансформировать в такой:

```
#include "winmain1.cpp"
void DrawStudyExample(HWND hWnd)
{
HDC hdc;

hdc = GetDC(hWnd);
if (hdc)
   for (int i=0; i<10; i++)
       Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
ReleaseDC(hWnd,hdc);
}
```

Это значительно короче. Большой кусок программы, включающий функции создания и обслуживания окна, спрятан в отдельных файлах, ссылка на которые выполнена директивой `#include`. Такая модульность с использованием отдельных кусков текста, которые вначале собираются в единый текст и затем компилируются все вместе, может эффективно применяться лишь для небольших программ, которые мы и рассмотрим в качестве примеров. Для более сложных программ лучше использовать проекты из раздельно компилируемых модулей OBJ, а также библиотеки DLL и компоненты типа ActiveX.

6.3. Использование графических функций API Windows

Операционная система Windows предоставляет программистам возможность использовать в своих программах сотни разнообразных функций, доступ к которым сделан в виде интерфейса **API** (Application Program Interface). При разработке программы на основе компьютерных языков, таких как C, C++, Pascal, в текст программы можно включать вызовы функций, которые входят в состав API Windows. После компиляции создается выполняемый файл (*.exe), который можно запускать на разных компьютерах с различными версиями ОС Windows, и программа будет корректно выполняться. Поскольку сами функции располагаются в модулях операционной системы, а в нашей программе содержатся только вызовы функций (call), то код выполняемого файла имеет небольшой размер.

Необходимо заметить, что мы будем рассматривать программы согласно спецификации Win32 для ОС Windows версий 95, 98, частично для NT и 2000. Существует несколько систем программирования, которые поддерживают разработку таких программ, например, Borland C++ [11, 59].

6.4. Контекст графического устройства

Графические функции из состава API Windows объединены в отдельную группу — подсистему **GDI** (Graphic Device Interface). Важная черта подсистемы GDI — аппаратная независимость многих функций от конкретного графического устройства.

Контекст графического устройства (Device Context) — это важный элемент графики в среде операционной системы Windows. Понятие контекста введено для описания того, где будет рисоваться изображение. Другими словами, контекст графического устройства указывает плоскость отображения, на которую делается графический вывод. В качестве контекста может быть окно программы на экране дисплея или страница принтера, или другое место, куда можно направить графический вывод.

Если ваша программа делает вызов графических функций API Windows, таких как рисование точек, линий, фигур, текста и тому подобных, необходимо указывать идентификатор контекста (handle of device context) и координаты. Вызов необходимого драйвера — для экрана дисплея, принтера или другого устройства — делает уже сама Windows. Это в значительной мере освобождает программиста от второстепенных дел и облегчает разработку программ, однако желательно учитывать специфику работы конкретного устройства.

Идентификатор контекста графического устройства (**hdc**) — это числовое значение, знание которого дает возможность направить графический вывод в нужное место. Перед началом рисования необходимо получить это числовое значение. После рисования обычно нужно освободить, деактивизировать контекст. Несоблюдение этого требования чревато неприятными последствиями — от утечек памяти до прекращения нормальной работы программы. Корректное использование контекста графического устройства осуществляется в такой последовательности:

1. Создание, активизация контекста, получение значения **hdc**.
2. Рисование с помощью графических функций API Windows.
3. Уничтожение, деактивизация контекста соответствующего **hdc**.

Контекст окна на экране дисплея

Для рисования в окне программы на экране дисплея можно использовать два способа. Эти способы различаются как по особенностям получения значения **hdc**, так и по возможностям рисования.

Первый способ основывается на использовании пары функций `GetDC` и `ReleaseDC`.

```
HDC hdc;
hdc = GetDC(hWnd);
. . . . .          //здесь вызовы функций рисования
. . . . .
ReleaseDC(hWnd,hdc);
```

Функция `GetDC` получает **hdc** для окна, заданного кодом `hWnd`. Например, для главного окна программы. Использование контекста графического вывода завершается вызовом функции `ReleaseDC`. Другую функцию для освобождения контекста в этом случае использовать не следует. Такой способ работы с контекстом использован в приведенном выше примере программы. Вообще этот способ можно рекомендовать везде, за исключением рисования во время обработки сообщения `WM_PAINT`.

Второй способ. Используется исключительно в теле обработчика сообщения `WM_PAINT` оконной функции.

```
PAINTSTRUCT ps;
HDC hdc;
hdc = BeginPaint(hWnd, &ps);
. . . . .                          //рисование
. . . . .
EndPaint(hWnd, &ps);
```

Во время обработки сообщения `WM_PAINT` функция `BeginPaint` обязательно должна вызываться первой, а `EndPaint` — последней.

Когда необходимо обрабатывать сообщения `WM_PAINT`? Это сообщение присылается любой программе тогда, когда повреждено изображение клиентской области окна этой программы. Повреждение изображения окна случаются довольно часто, например, при отображении на экране нескольких окон — когда на окно было наложено еще одно окно, а потом после закрытия последнего окна части изображения первого окна уже нет. Если в программе предусмотрена перерисовка изображения рабочей (клиентской) области окна

путем программирования обработчика сообщения WM_PAINT, то это позволяет в большинстве случаев гарантировать корректное отображение окна.

Приведем пример программы, рисующей согласно второму способу.

```
#include "winmain.cpp"
void DrawStudyExample(HWND hWnd);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
  {
  case WM_PAINT:
    DrawStudyExample(hWnd);
    break;
  case WM_COMMAND:
    switch (LOWORD(wParam))
      {
      case 108:
        DestroyWindow(hWnd);
        break;
      default : break;
      }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:return DefWindowProc(hWnd,message,wParam,lParam);
  }
return 0L;
}

void DrawStudyExample(HWND hWnd)
{
HDC hdc;
PAINTSTRUCT ps;
hdc = BeginPaint(hWnd, &ps);
for (int i=0; i<10; i++)
    Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
EndPaint(hWnd, &ps);
}
```

Контекст принтера

Рассмотрим, как можно нарисовать (а точнее, напечатать) что-то на принтере. Для этого также сначала необходимо получить значение **hdc** соответствующего графического устройства. Потом выполняется рисование. А затем контекст освобождается. Общая схема такая же. Пример программы:

```
#include "winmain.cpp"
void DrawStudyExample(HWND hWnd);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
  {
  case WM_COMMAND:
    switch (LOWORD(wParam))
      {
      case 103:                  //выбран пункт меню Печать
        DrawStudyExample(hWnd);
        break;
      case 108:
        DestroyWindow(hWnd);
        break;
      default : break;
      }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:return DefWindowProc(hWnd,message,wParam,lParam);
  }
return 0L;
}
void DrawStudyExample(HWND hWnd)
{
HDC hdc;
PRINTER_INFO_5 pinfo5[3];
DWORD dwNeeded, dwReturned;
DOCINFO di = {sizeof (DOCINFO), "StudEx", NULL};
if (!EnumPrinters (PRINTER_ENUM_DEFAULT,
                   NULL,
                   5,
                   (LPBYTE) pinfo5,
```

```
                    sizeof (pinfo5),
                    &dwNeeded,
                    &dwReturned)) return;
if (! pinfo5[0]. pPrinterName) return;
hdc = CreateDC (NULL,
                pinfo5[0]. pPrinterName,    //имя принтера
                NULL, NULL);
if (hdc == NULL) return;      //ошибка создания контекста
if (StartDoc(hdc, &di) > 0)
   {
   if (StartPage(hdc) > 0)
      {
      for (int i=0; i<10; i++)
          Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
      EndPage(hdc);
      }
   EndDoc(hdc);
   }
DeleteDC(hdc);
}
```

Для принтера значение **hdc** получить несколько сложнее. Сначала необходимо узнать имя принтера — здесь это делается с помощью функции `EnumPrinters`. Обратите внимание на то, что использована не одна структура, а массив из трех структур `pinfo5` — это сделано согласно рекомендациям [20]. Потом создается контекст вывода с помощью функции `CreateDC`. Печать на принтере выполняется с помощью функций `StartDoc`, `StartPage`, `EndPage` и `EndDoc`. После вызова функции `EndDoc` контекст принтера необходимо освободить с помощью функции `DeleteDC`.

Следует заметить, что приведенный пример предназначен для Windows 95, 98. Для Windows NT функцию `DrawStudyExample` необходимо модифицировать — об этом можно узнать в документации Win32 SDK [61].

После печати на бумаге можно заметить отличия результата от изображения на экране. Прежде всего, это касается размера прямоугольников. Для лазерного принтера размеры значительно меньше, а для матричного размеры могут быть и больше. Это объясняется различными разрешающими способностями (dpi) принтеров и экрана дисплея. Кроме того, на матричном принтере рисунок может изменить пропорции — в случае, когда (dpi) по вертикали отличается от (dpi) по горизонтали. При вызове функции `Rectangle` мы использовали координаты в виде пикселов. Функции API Windows разрешают использовать и другие системы координат.

Контекст метафайла

Рассмотрим еще одну разновидность контекста — контекст метафайла. Здесь рисунок создается в виде файла на диске. Известны два формата метафайлов для Windows: WMF и EMF. Формат EMF более совершенный. В качестве примера приведем текст программы, записывающей рисунок в файл *.emf после выбора пункта меню "Записать Как".

```
#include "winmain.cpp"
void DrawStudyExample(HWND hWnd);

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
  {
  case WM_COMMAND:
    switch (LOWORD(wParam))
      {
      case 102:          //выбран пункт меню Записать Как
        DrawStudyExample(hWnd);
        break;
      case 108:
        DestroyWindow(hWnd);
        break;

      default : break;
      }
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:return DefWindowProc(hWnd,message,wParam,lParam);
  }
return 0L;
}
#include <mem.h>
void DrawStudyExample(HWND hWnd)
{
//------сначала используем стандартное окно SaveAs-------
//------для определения полного имени файла (szFname)----
OPENFILENAME ofn;
char szFname[256];
```

```
szFname[0] = 0;
memset(&ofn, 0, sizeof (OPENFILENAME));
ofn.lStructSize = sizeof (OPENFILENAME);
ofn.hwndOwner   = hWnd;
ofn.lpstrFilter = "Метафайлы (*.emf)\0*.emf\0\0",
ofn.lpstrDefExt = "emf";
ofn.lpstrFile   = szFname;
ofn. nMaxFile   = 256;
if (!GetSaveFileName(&ofn)) return;
//--------------теперь создание метафайла-----------------
HDC hdc;
HENHMETAFILE hemf;
hdc = CreateEnhMetaFile(NULL, szFname ,NULL,NULL);
if (hdc == NULL) return;  //ошибка, контекст не создан
for (int i=0; i<10; i++)
    Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
hemf = CloseEnhMetaFile(hdc);  //запись на диск и закрытие
//--демонстрация возможностей отображения метафайла в окне
HDC hdcWin;
RECT rc;
hdcWin = GetDC(hWnd);              //берем контекст окна
for (int i=0;i<4;i++)       //16 раз отобразим метафайл
    for (int j=0;j<4;j++)  //в ячейках 45x45 пикселов
        {
        rc. left  = 50*i;    //текущие границы отображения
        rc.top    = 50*j;
        rc.right  = 45 + rc.left;
        rc.bottom = 45 + rc.top;
        PlayEnhMetaFile(hdcWin,hemf, &rc);
        }
//------еще одно отображение метафайла (крестиком)------
rc.left = 210;
rc.top = 20;
rc.right = 380;
rc. bottom = 180;
PlayEnhMetaFile(hdcWin,hemf, &rc);
rc. top    = 180;       //а здесь зеркальный поворот
rc. bottom = 20;
PlayEnhMetaFile(hdcWin,hemf, &rc);
ReleaseDC(hWnd,hdcWin);         //освобождаем контекст окна,
DeleteEnhMetaFile(hemf);        //освобождаем память,
}                               //а файл на диске остается
```

Запустите программу и выберите меню "Файл \ Записать как". После определения имени файла и записи метафайла на диск в окне должно появиться следующее изображение (рис. 6.2).

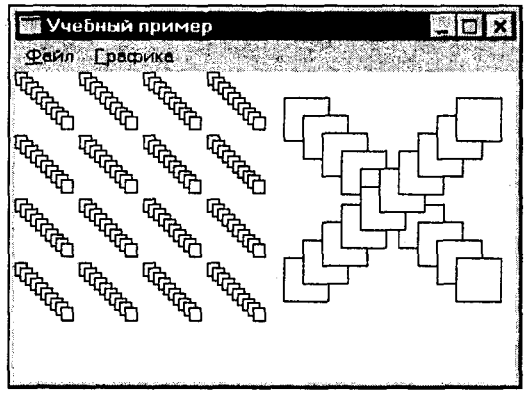


Рис. 6.2. Пример метафайла

Метафайл описывает изображение в виде последовательности вызовов графических функций, использованных при создании изображения (в приведенном примере программы это 10 вызовов функции `Rectangle`). В метафайл записываются функции вместе со значениями аргументов-координат, границы рисунка и другая информация. Общие параметры рисунка можно узнать, если прочитать заголовок метафайла с помощью функции `GetEnhMetaFileHeader`.

Для отображения метафайла нужно задать границы вывода. Поскольку метафайл — векторное описание изображения, то это дает возможность при отображении плавно растягивать рисунок, причем намного лучше, чем растровые изображения. В нашей программе метафайл отображен 18 раз в контексте главного окна. Обратите внимание на крестик — здесь использовано зеркальное отображение.

В результате работы этой программы на диске должен появиться файл *.emf. Чтобы проверить содержимое файла, вызывайте любую программу, которая может читать файлы формата EMF — например, Word для Windows. Выполните вставку рисунка *.emf и вы увидите десять прямоугольников. Рисунок векторный, поэтому его можно свободно растягивать. Границы рисунка определяются согласно диапазону координат всех элементов рисунка — аргументов вызова функции `Rectangle`.

Необходимо заметить, что нижний прямоугольник этого рисунка обычно выглядит поврежденным (как именно — это еще зависит от версии Word).

Этот прямоугольник рисуется согласно координатам при $i = 9$: `Rectangle(hdc, 9*30, 9*20, 9*30+70, 9*20+50)`. Координаты правого нижнего угла (`9*30+70, 9*20+50`) этого прямоугольника определяют правую и нижнюю границу всего рисунка. Границы рисунка записываются в метафайл в виде координат левого верхнего и правого нижнего углов. Чтобы узнать, какие границы определены для метафайла, можно использовать функцию `GetEnhMetaFileHeader`:

```
ENHMETAHEADER emh;
GetEnhMetaFileHeader(hemf, sizeof(ENHMETAHEADER), &emh);
```

Эта функция заполняет поля структуры `emh`, имеющей тип `ENHMETAHEADER`. Границы рисунка записываются в поля `emh.rclBounds`, и имеют такие значения:

```
emh.rclBounds.left = 0;
emh.rclBounds.top = 0;
emh. rclBounds. right  = 339;
emh. rclBounds. bottom = 229;
```

Обратите внимание на то, что координаты правого нижнего угла имеют значения на единицу меньше, чем должны быть. Это — проявление особенностей функции `Rectangle`, рисующей правый нижний угол контура как раз на один пиксел ближе к левому верхнему углу. К тому можно привыкнуть, но почему Microsoft Word (версий 6.0 для 95 и 97-й) искажает рисунок (отрезает границы крайнего прямоугольника)? Средствами Word можно расширить границы этого рисунка, но для других файлов *. emf это может привести, например, к повреждению стиля линий. Поэтому попробуем расширить границы рисунка собственноручно, например, так:

```
SetPixel(hdc, 9*30+70, 9*20+50, RGB(255,255,255));
for (int i=0; i<10; i++)
    Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
```

Здесь мы сначала рисуем один белый пиксел с помощью функции `SetPixel`, что никак не изменяет сам рисунок, но координаты его `(x,y) = (9*30+70, 9*20+50) = (340, 230)` будут учтены при определении границ рисунка. Запустите модифицированную программу, запишите метафайл, и попробуйте загрузить этот метафайл в редактор Word. Рисунок должен выглядеть неповрежденным.

Кроме того, в некоторых рисунках, которые записываются в метафайлы, необходимо учитывать толщину пера. В нашем случае используется перо по

умолчанию, имеющее толщину в один пиксел. Если рисовать толстыми перьями, то это также может привести к ошибкам определения границ рисунка — не только для правого нижнего, а и левого верхнего углов. Рассмотрим такой пример:

```
HPEN hPenOld,hPen;
SetPixel(hdc, -1, -1, RGB(255,255,255));
SetPixel(hdc, 9*30+71, 9*20+51, RGB(255,255,255));
hPen = CreatePen(PS_SOLID, 3, RGB(0,0,0));//толщина пера
                                          //три пиксела
hPenOld = (HPEN)SelectObject(hdc, hPen);
for (int i=0; i<10; i++)
    Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
SelectObject(hdc, hPenOld);
DeleteObject(hPen);
```

В приведенном примере две функции `SetPixel` использованы для установления границ рисунка (–1, –1, 341, 231). Если выбросить эти две функции, то границы будут (0, 0, 339, 229), что может привести к проблемам с использованием метафайла в программе Word.

Необходимо отметить, что в нашей программе функция API `PlayEnhMetaFile` корректно отображает этот метафайл, поэтому к ней никаких претензий, а расширение границ с помощью функции `SetPixel` — это попытка исправить ошибки соответствующих прикладных программ.

Если при создании метафайла его имя отсутствует, например `CreateEnhMetaFile(NULL,NULL,NULL,NULL)`, то метафайл не будет записываться на диск, поверхность контекста будет существовать только в памяти компьютера. Такой метафайл также можно использовать, например, отображать в другом контексте с помощью функции `PlayEnhMetaFile`.

Контекст памяти

Рассмотрим контекст памяти (memory context), для которого поверхность рисования создается на основе битовой карты (bitmap).

```
#include "winmain1.cpp"
void DrawStudyExample(HWND hWnd)
{
HDC hdc,hdcWin;
HBITMAP hBitmap;
hdcWin = GetDC(hWnd);
hdc = CreateCompatibleDC(hdcWin);
```

```
hBitmap = CreateCompatibleBitmap(hdcWin, 400, 300);
SelectObject(hdc,hBitmap);
Rectangle(hdc, -1, -1, 401, 301);      //очистка поверхности
for (int i=0; i<10; i++)
    Rectangle(hdc, i*30, i*20, i*30+70, i*20+50);
BitBlt(hdcWin,0,0,400,300, hdc,0,0,SRCCOPY);  //копирование
DeleteDC(hdc);
DeleteObject(hBitmap);
ReleaseDC(hWnd,hdcWin);
}
```

Контекст в памяти создается вызовом функции `CreateCompatibleDC`. Здесь он создается по образцу контекста окна программы. Но в таком контексте отсутствует поверхность рисования. Эту поверхность создаем по образцу цветового формата поверхности окна экрана вызовом функций `CreateCompatibleBitmap` и `SelectObject`. Потом можно рисовать в соответствии с `hdc`. Сначала мы очищаем поверхность (закрашиваем белым цветом), а потом рисуем десять прямоугольников. Потом вызов функции `BitBlt`, копирующей растровое изображение из контекста памяти (`hdc`) в контекст окна (`hdcWin`). После использования контекста памяти его необходимо закрыть и освободить выделенную память, которая была выделена. Здесь это сделано функциями `DeleteDC` и `DeleteObject`.

Контекст памяти часто используется для "заэкранного" рисования — например, для анимации. Рассмотрим один из простейших примеров анимации — в цикле рисуется по десять прямоугольников, размеры которых изменяются от 5 до 300. Это мы сделаем на основе текста программы для предыдущих примеров.

```
//-----------попытка анимации - растущие прямоугольники-----
#include "winmain1.cpp"
void DrawStudyExample(HWND hWnd)
{
HDC hdc;
hdc = GetDC(hWnd);
for (int size=5; size<=300; size++)
    {
    Rectangle(hdc, -1, -1, 401, 301);
    for (int i=0; i<10; i++)
        Rectangle(hdc, i*30, i*20, i*30+size, i*20+size);
    }
ReleaseDC(hWnd,hdc);
}
```

Изображение кажется подвижным, но поскольку здесь все рисуется непосредственно в окне программы, то это воспринимается плохо. Сравните с работой следующего примера, где использован контекст памяти.

```
//----------------более совершенная анимация---------------
#include "winmain1.cpp"
void DrawStudyExample(HWND hWnd)
{
HDC hdc,hdcWin;
HBITMAP hBitmap;
hdcWin = GetDC(hWnd);
hdc = CreateCompatibleDC(hdcWin);
hBitmap = CreateCompatibleBitmap(hdcWin, 400, 300);
SelectObject(hdc,hBitmap);
for (int size=5; size<=300; size++)
    {
    Rectangle(hdc, -1, -1, 401, 301);
    for (int i=0; i<10; i++)
        Rectangle(hdc, i*30, i*20, i*30+size, i*20+size);
    BitBlt(hdcWin,0,0,400,300, hdc,0,0,SRCCOPY);
    }
DeleteDC(hdc);
DeleteObject(hBitmap);
ReleaseDC(hWnd,hdcWin);
}
```

Изображение вначале рисуется в контексте памяти, а затем быстро копируется на экран с помощью функции `BitBlt`.

Программы КГ часто применяют подобный способ рисования — "за экраном". Этот способ называется ***двойной буферизацией*** (double buffering). Кроме анимации, использование двойной буферизации иногда позволяет ускорить рисование — если графические функции могут рисовать в контексте памяти быстрее, чем в окне на экране. Также полезно использование контекста памяти тогда, когда программа, например графический редактор, манипулирует изображением, которое по размерам больше, чем способен отобразить экран дисплея.

Параметры контекста графического устройства

Для рисования важно знать параметры графического устройства, такие как размеры области отображения, количество цветов. Для этого можно использовать функцию `GetDeviceCaps`:

```
value = GetDeviceCaps (hdc, index);
```

Для того чтобы узнать необходимые параметры контекста hdc, нужно в качестве аргумента index задавать соответствующие значения. Например:

```
cx   = GetDeviceCaps (hdc, HORZRES);
cy   = GetDeviceCaps (hdc, VERTRES);
dpiX = GetDeviceCaps (hdc, LOGPIXELSX);
dpiY = GetDeviceCaps (hdc, LOGPIXELSY);
bits = GetDeviceCaps (hdc, BITSPIXEL);
```

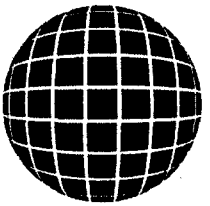
Так можно узнать размеры в пикселах (cx, cy), разрешающую способность (dpiX, dpiY), количество бит на пиксел для цвета (bits) и много других параметров.

Такие параметры, как размеры изображения (cx, cy) для окна можно получить и другими способами, например, вызовом функции GetClientRect:

```
RECT rc;
GetClientRect(hWnd, &rc);
```

Эта функция заполняет поля структуры rc типа RECT. Поля rc.left и rc.top заполняются нулями, а поля rc.right и rc.bottom хранят координаты соответственно правого и нижнего края данного окна.

ГЛАВА 7



Графические примитивы API Windows

7.1. Отдельные пикселы

Функция `SetPixel` рисует один пиксел растра. Она имеет такие аргументы:

```
SetPixel(hdc, x, y, clr),
```

где `hdc` — контекст, `x,y` — координаты, `clr` — цвет пиксела. Аргумент `clr` имеет тип 4-байтного `COLORREF`, причем три младших байта соответствуют компонентам `Blue,Green,Red` (в диапазоне от 0 до 255 каждая), а старший байт не используется. Цвет, кодируемый типом `COLORREF`, удобно задавать макросом `RGB(r,g,b)`. Функция `SetPixel` сама имеет тип `COLORREF` — она возвращает значение цвета пиксела.

Кроме `SetPixel` в API Win32 есть функция `SetPixelV`, работающая немного быстрее, поскольку не возвращает значения; а также предусмотрена функция для получения цвета любого пиксела растра — `GetPixel`.

С помощью таких функций, которые оперируют отдельными пикселами, очевидно, возможно создать любое растровое изображение. Рассмотрим, как нарисовать шар. Пусть источник света расположен далеко позади нас, а шар отражает свет согласно модели диффузного рассеяния пропорционально квадрату косинуса угла нормали. Рисование шара запрограммируем в виде функции `MySphere`. Текст программы `studex6.cpp`.

```
//-------(c)Copyright Порев В.Н.----------
#include "winmain1.cpp"
void MySphere(HDC hdc,int xc,int yc,int R,
              BYTE red,BYTE gre,BYTE blu);
```

```
void DrawStudyExample(HWND hWnd)
{
HDC hdc;
RECT rc;
int xc,yc,x,y,r,rmax,d;
hdc = GetDC(hWnd);
if (hdc == NULL) return;
GetClientRect(hWnd,&rc);
Rectangle(hdc,0,0,rc.right,rc.bottom);
xc = (rc.right-rc.left)/2;
yc = (rc.bottom-rc.top)/2;
if (xc <= yc)
     rmax = xc/3;
else rmax = yc/3;
MySphere(hdc, xc, yc, rmax, 255,255,0);
d = 0;
for (int i=1; i<=3; i++)
    {
    d += rmax/i + rmax/(i+1);
    x = (double)d*0.866;
    y = (double)d*0.5;
    r = rmax/(i+1);
    if (i == 1) MySphere(hdc, xc, yc-d, r, 255,0,0);
    if (i <= 2)
       {
       MySphere(hdc, xc+x, yc-y, r, 0,255,255);
       MySphere(hdc, xc-x, yc-y, r, 0,255,255);
       }
    MySphere(hdc, xc+y, yc+x, r, 0,255,0);
    MySphere(hdc, xc-y, yc+x, r, 0,255,0);
    }
ReleaseDC(hWnd,hdc);
}

void MySphere(HDC hdc,int xc,int yc,int R,
              BYTE red,BYTE gre,BYTE blu)
{
COLORREF clr;
double R2,r2,k;
int x,y;

R2 = R*R;
for (x=0;x<=R;x++)
```

```
  for (y=0;y<=x;y++)          //для одного октанта
      {
      r2 = x*x + y*y;
      if (r2 > R2) break;
      k = 1 - r2/R2;          //k = cos² (угла нормали)
      clr = RGB((BYTE)(k*(double)red),
                (BYTE)(k*(double)gre),
                (BYTE)(k*(double)blu));
      SetPixel(hdc,xc+x,yc+y,clr);
      SetPixel(hdc,xc+x,yc-y,clr);
      SetPixel(hdc,xc-x,yc+y,clr);
      SetPixel(hdc,xc-x,yc-y,clr);
      SetPixel(hdc,xc+y,yc+x,clr);
      SetPixel(hdc,xc+y,yc-x,clr);
      SetPixel(hdc,xc-y,yc+x,clr);    //используем
      SetPixel(hdc,xc-y,yc-x,clr);    //симметрию шара
      }
}
```

Получим рисунок, подобный рис. 7.1. В особенности обратите внимание на то, что для качественного отображения такого рисунка необходимо установить, как минимум, 16-битный (а лучше 24- или 32-битный) видеорежим дисплея. Для 256-цветного видеорежима изображение будет плохим — много оттенков будет искажено.

Приведенный выше пример программы можно рассматривать как попытку имитации изображения объемных объектов. Эта программа имеет очень ограниченные возможности. Например, если мы попробуем сдвинуть шары, то получится так, как на рис. 7.2. Это тоже имитация объема — маленькие шары выглядят более близкими, поскольку мы их нарисовали в последнюю очередь. А как получить изображение шаров, которые частично входят вглубь один другого (рис. 7.3)?

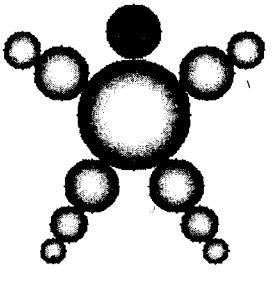


Рис. 7.1. Первый пример

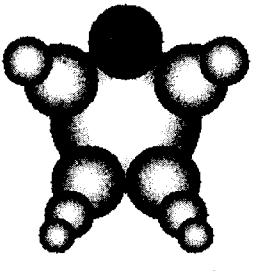


Рис. 7.2. Второй пример

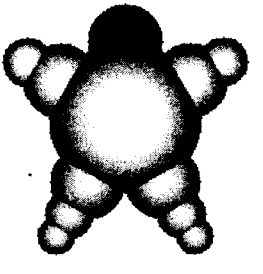


Рис. 7.3. Третий пример

Любая последовательность рисования отдельных шаров не даст желаемого результата. Необходимо как-то отрезать части шаров. Для создания того, что нам нужно, используем универсальный метод — метод Z-буфера. Рассмотрим текст программы studex7.cpp.

```
//-------------(c)Copyright Порев В.Н.-------------------
#include "winmain1.cpp"
float *pZbuf = NULL;   //глобальные переменные Z-буфера
long horSize;

void MySphereZ(HDC hdc,int xc,int yc,int R,
               BYTE red,BYTE gre,BYTE blu);
BOOL InitMyZbuffer(long cx, long cy);
void SetPixMyZ(HDC hdc,int x,int y,float z,COLORREF clr);

void DrawStudyExample(HWND hWnd)
{
HDC hdc;
RECT rc;
int xc,yc,x,y,r,rmax,d;

hdc = GetDC(hWnd);
if (hdc == NULL) return;
GetClientRect(hWnd,&rc);
if (InitMyZbuffer(rc.right,rc.bottom))  //создаем Z-буфер
   {
   Rectangle(hdc,0,0,rc.right,rc.bottom);
   xc = (rc.right-rc.left)/2;
   yc = (rc.bottom-rc.top)/2;
   if (xc <= yc)
        rmax = xc/3;
   else rmax = yc/3;
   MySphereZ(hdc, xc, yc, rmax, 255,255,0);
   d = 0;
   for (int i=1; i<=3; i++)
       {
       d += rmax/i;
       x = (double)d*0.866;
       y = (double)d*0.5;
       r = rmax/(i+1);
       if (i == 1) MySphereZ(hdc,xc,yc-d,r,255,0,0);
       if (i <= 2)
```

```
        {
        MySphereZ(hdc, xc+x, yc-y, r, 0,255,255);
        MySphereZ(hdc, xc-x, yc-y, r, 0,255,255);
        }
      MySphereZ(hdc, xc+y, yc+x, r, 0,255,0);
      MySphereZ(hdc, xc-y, yc+x, r, 0,255,0);
      }
   delete []pZbuf;              //уничтожаем Z-буфер
   }
ReleaseDC(hWnd,hdc);
}
//-------модифицированная функция рисования шара---------
void MySphereZ(HDC hdc,int xc,int yc,int R,
               BYTE red,BYTE gre,BYTE blu)
{
COLORREF clr;
double R2,r2,z2,k;
int x,y;

R2 = R*R;
for (x=0;x<=R;x++)
  for (y=0;y<=x;y++)
      {
      r2 = x*x + y*y;
      if (r2 > R2) break;
      z2 = R2 - r2;        //квадрат расстояния (z^2)
      k = 1 - r2/R2;
      clr = RGB((BYTE)(k*(double)red),
                (BYTE)(k*(double)gre),
                (BYTE)(k*(double)blu));
      SetPixMyZ(hdc,xc+x,yc+y,z2,clr);
      SetPixMyZ(hdc,xc+x,yc-y,z2,clr);
      SetPixMyZ(hdc,xc-x,yc+y,z2,clr);
      SetPixMyZ(hdc,xc-x,yc-y,z2,clr);
      SetPixMyZ(hdc,xc+y,yc+x,z2,clr);
      SetPixMyZ(hdc,xc+y,yc-x,z2,clr);
      SetPixMyZ(hdc,xc-y,yc+x,z2,clr);
      SetPixMyZ(hdc,xc-y,yc-x,z2,clr);
      }
}
//--------начальная инициализация Z-буфера-----------
BOOL InitMyZbuffer(long cx, long cy)
{
long i,size;
```

```
size = cx * cy;
pZbuf = new float [size];
if (pZbuf == NULL) return FALSE;
horSize  = cx;
for (i=0; i<size; i++)
    pZbuf [i] = -10000;
return TRUE;
}
//------рисование одного пиксела с проверкой по z-----
void SetPixMyZ(HDC hdc,int x,int y,float z,COLORREF clr)
{
long indx;
indx = (long)y*horSize + (long)x;
if (z >= pZbuf [indx])          //этот пиксел ближе
   {
   pZbuf [indx] = z;
   SetPixel(hdc,x,y,clr);
   }
}
```

В этом примере для удаления невидимых точек реализован метод Z-буфера. Однако в качества меры дальности (глубины) используется не z, а z^2 для упрощения вычислений — не нужно определять квадратный корень. Это вполне корректно, поскольку отрицательные значения z здесь не встречаются. Удаление невидимых точек реализовано функцией `SetPixMyZ`. Если текущее значение (z^2) оказывается большим, чем содержимое Z-буфера, то пиксел рисуется в растре, а значение z^2 записывается в Z-буфер. Необходимо заметить, что для уменьшения затрат памяти можно открывать Z-буфер не для всего окна, а только в границах рисунка.

Подвижные шары

В следующем примере использования функции `SetPixel` рассмотрим некоторые аспекты расчета трехмерных координат. В качестве пространственных объектов снова используем шары. Попробуем моделировать движение шаров, которые вращаются один вокруг другого (рис. 7.4).

Положение шаров зададим с помощью углов α и β. Эти углы описывают положения в локальных системах координат. Здесь определены три системы координат — две локальные для отдельных шаров и система координат (X, Y, Z) для окна программы. Для рисования шаров необходимо указывать оконные координаты. Чтобы вычислять эти координаты, используются мат-

рицы коэффициентов (в тексте программы матрицы имитируются одномерными массивами). Матрица `matrix1` отвечает преобразованию координат из системы ($x1$, $y1$, $z1$) в (X, Y, Z). Матрица `matrix2` — для преобразования ($x2$, $y2$, $z2$) в (X, Y, Z). Кроме того, указанные преобразования координат обеспечивают центрирование изображения в окне программы.

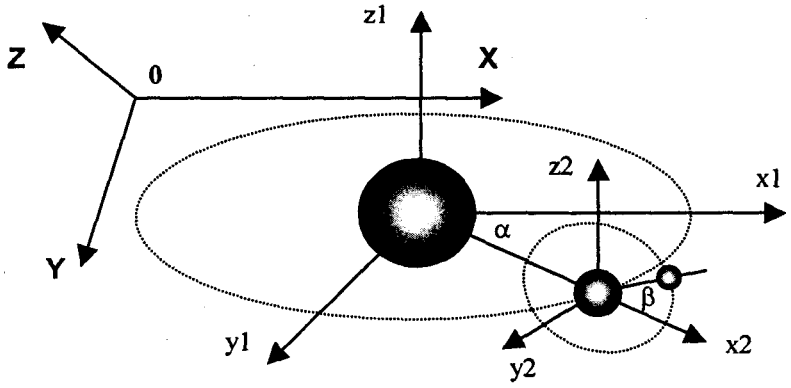


Рис. 7.4. Движение шаров

Для имитации движения использован цикл, в котором изменяются углы α и β. Цикл движения составляет один полный оборот среднего шара. Угол β изменяется быстрее, поэтому маленький шар делает несколько оборотов.

Рассмотрим текст программы `studex8.cpp`.

```
//----------(c)Copyright Порев В.Н.------------
#include "winmain1.cpp"
#include <math.h>
#include <mem.h>

void MyTransformCoords(double *X, double *Y, double *Z,
                       double  x, double  y, double  z,
                       double *koef);
void SetShiftMatrix(double *matrix,
                    double dx,double dy,double dz);
void SetXRotateMatrix(double *matrix, double a);
void SetZRotateMatrix(double *matrix, double a);
void MatrixAxB(double *matrix,double *A, double *B);
void MySphere(HDC hdc,int xc,int yc,int R,
              BYTE red,BYTE gre,BYTE blu);
```

```
void DrawStudyExample(HWND hWnd)
{
HDC  hdc,hdcWin;
HBITMAP hBitmap;
RECT rc;

const int numSteps=360;
int traceX2[numSteps],traceY2[numSteps],
    traceX3[numSteps],traceY3[numSteps],num=0;

double Alpha,Beta,stepA,stepB;
double x1,y1,x2,z2;
double r1,r2,r3,R2,R3;
double X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3;
double matrix1[16],matrix2[16],
       matrixA[16],matrixB[16];

GetClientRect(hWnd,&rc);
hdcWin = GetDC(hWnd);
hdc = CreateCompatibleDC(hdcWin);
hBitmap = CreateCompatibleBitmap(hdcWin,
                                 rc.right,rc.bottom);
SelectObject(hdc,hBitmap);
R3 = rc.right/12;            //радиусы орбит
R2 = rc.right/2 - 2*R3;
r1 = (R2-R3)/4;              //радиусы шаров
r2 = r1/2;
r3 = r2/2;
SetXRotateMatrix(matrixA, -M_PI/3);
SetShiftMatrix(matrixB, -rc.right/2,-rc.bottom/2,0);
MatrixAxB(matrix1,matrixB, matrixA);
Alpha = Beta = 0;
stepA = 2*M_PI/numSteps;
stepB = 10*stepA;
for (int i=0; i<=numSteps; i++)
    {
    Rectangle(hdc,0,0,rc.right,rc.bottom);
    MyTransformCoords(&X1,&Y1,&Z1,
                      0,  0,  0,
                     matrix1);
    MySphere(hdc, X1,Y1,r1, 255,255,0);
    x1 = R2*cos(Alpha);
    y1 = R2*sin(Alpha);
```

```
    MyTransformCoords(&X2,&Y2,&Z2,
                      x1, y1,  0,
                     matrix1);
    x2 = R3*cos(Beta);
    z2 = R3*sin(Beta);
    SetZRotateMatrix(matrixA, -Alpha);
    SetShiftMatrix(matrixB, -x1,-y1,0);
    MatrixAxB(matrixB, matrixB, matrixA);
    MatrixAxB(matrix2, matrix1, matrixB);
    MyTransformCoords(&X3,&Y3,&Z3,
                      x2,  0, z2,
                     matrix2);
    traceX2[num] = X2;
    traceY2[num] = Y2;
    traceX3[num] = X3;
    traceY3[num] = Y3;
    num++;
    for (int j=0; j<num; j++)
        {
        SetPixel(hdc,traceX2[j],traceY2[j],RGB(0,0,0));
        SetPixel(hdc,traceX3[j],traceY3[j],RGB(0,0,0));
        }
    if (Z2 <= Z3)
       {
       MySphere(hdc, X2,Y2,r2, 0,255,0);
       MySphere(hdc, X3,Y3,r3, 255,0,0);
       }
    else
       {
       MySphere(hdc, X3,Y3,r3, 255,0,0);
       MySphere(hdc, X2,Y2,r2, 0,255,0);
       }
    BitBlt(hdcWin,0,0,rc.right,rc.bottom, hdc,0,0,SRCCOPY);
    Alpha += stepA;
    Beta  += stepB;
    }
DeleteDC(hdc);
DeleteObject(hBitmap);
ReleaseDC(hWnd,hdcWin);
}
//---преобразование координат, заданное матрицей koef[]--
void MyTransformCoords(double *X, double *Y, double *Z,
                       double  x, double  y, double  z,
                       double *koef)
```

```
{
*X = koef[0]*x + koef[1]*y + koef[2]*z  +  koef[3];
*Y = koef[4]*x + koef[5]*y + koef[6]*z  +  koef[7];
*Z = koef[8]*x + koef[9]*y + koef[10]*z + koef[11];
}
//----------формирование матрицы сдвига------------
void SetShiftMatrix(double *matrix,
                    double dx,double dy,double dz)
{
for (int i=0; i<16; i++) matrix[i]=0;
matrix[0]  = 1;
matrix[5]  = 1;
matrix[10] = 1;
matrix[15] = 1;
matrix[3]  = -dx;
matrix[7]  = -dy;
matrix[11] = -dz;
}
//-----формирование матрицы поворота вокруг оси X-------
void SetXRotateMatrix(double *matrix, double a)
{
for (int i=0; i<16; i++) matrix[i]=0;
matrix[0]  = 1;
matrix[15] = 1;
matrix[5]  = cos(a);
matrix[6]  = sin(a);
matrix[9]  = -sin(a);
matrix[10] = cos(a);
}
//-----формирование матрицы поворота вокруг оси Z-------
void SetZRotateMatrix(double *matrix, double a)
{
for (int i=0; i<16; i++) matrix[i]=0;
matrix[0]  = cos(a);
matrix[1]  = sin(a);
matrix[4]  = -sin(a);
matrix[5]  = cos(a);
matrix[10] = 1;
matrix[15] = 1;
}
//-----------------перемножение матриц-----------------
//-------матрицы задаются массивами A[16], B[16]--------
void MatrixAxB(double *matrix,double *A, double *B)
```

```
int indx,i,j,k;
double tmp[16];

for (indx=0; indx<16; indx++)
    {
    i = 4*(indx/4);
    j = indx % 4;
    tmp[indx]=0;
    for (k=0; k<4; k++)
        {
        tmp[indx] += A[i]*B[j];
        i++;
        j += 4;
        }
    }
memcpy(matrix,tmp,sizeof(tmp));
}

void MySphere(HDC hdc,int xc,int yc,int R,
              BYTE red,BYTE gre,BYTE blu)
{
. . . //текст этой функции полностью идентичен
. . . //тексту функции MySphere примера studex6
}
```

Запустите программу на выполнение, выберите пункт меню "Графика" и подождите, пока зеленый шар опишет один оборот. В результате получим изображение, как на рис. 7.5.

Необходимо заметить, что эту программу можно было бы упростить. Например, рисование эллипса трасы среднего шара лучше делать функцией `Arc`, а не использовать массивы для координат точек эллипса и рисовать каждую точку функцией `SetPixel`. Кроме того, не обязательно каждый раз рисовать центральный шар — это очень замедляет рисование (и вдобавок он — самый большой). Ускорить рисования можно, если рисование центрального шара вынести из общего цикла — нарисовать его лишь один раз, а в цикле стирать изображения только двух подвижных шаров.

Кроме того, в этом примере так подобран ракурс обзора, чтобы упростить программирование трехмерного показа. В первую очередь это касается удаления невидимых точек методом сортировки объектов по глубине (дальности). В программе использована простая сортировка по глубине для второго

и третьего шаров путем сравнения координат Z центров шаров. Центральный шар здесь рисуется всегда первым — но будет ошибкой, если увеличить угол наклона плоскости проецирования. Для решения этой проблемы можно сделать сортировку по глубине более универсальной — для всех трех шаров.

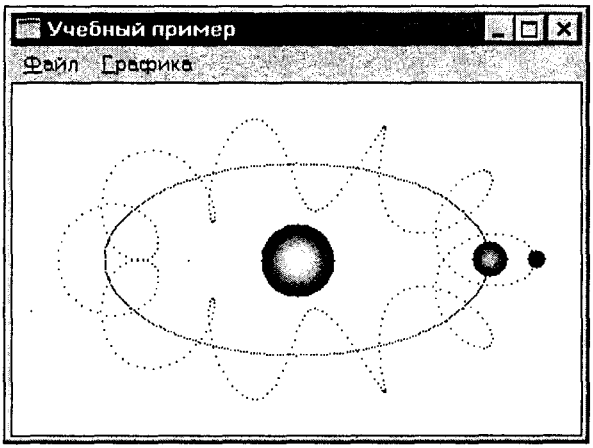


Рис. 7.5. Трассы движения шаров

Интересный рисунок можно получить, если использовать основные идеи программы `studex8` и предыдущей программы `studex7`. Используем Z-буфер для рисования спирали (рис. 7.6).

Рис. 7.6. Движение двух шаров

Текст соответствующей программы (`studex9`) дадим только для тех функций, которые отличаются от соответствующих функций программ `studex7,8`.

```
//----------(c)Copyright Порев В.Н.-----------
void DrawStudyExample(HWND hWnd)
{
HDC  hdc;
RECT rc;
const int numSteps=180;
double Alpha,Beta,stepA,stepB;
double x1,y1,x2,z2;
double r1,r2,R1,R2;
double X1,Y1,Z1,X2,Y2,Z2;
double matrix1[16],matrix2[16];
double matrixA[16],matrixB[16];

GetClientRect(hWnd,&rc);
hdc = GetDC(hWnd);
R2 = rc.right/10;
R1 = rc.right/2 - 2*R2;
r1 = (R1-R2)/4;
r2 = r1/2;
SetXRotateMatrix(matrixA, -M_PI/3);
SetShiftMatrix(matrixB, -rc.right/2,-rc.bottom/2,0);
MatrixAxB(matrix1,matrixB, matrixA);
Alpha = Beta = 0;
stepA = 2*M_PI/numSteps;
stepB = 10*stepA;
if (!InitMyZbuffer(rc.right,rc.bottom))
   {
   ReleaseDC(hWnd,hdc);
   return;
   }
Rectangle(hdc,0,0,rc.right,rc.bottom);
for (int i=0; i<=numSteps; i++)
  {
  x1 = R1*cos(Alpha);
  y1 = R1*sin(Alpha);
  MyTransformCoords(&X1,&Y1,&Z1,
                    x1, y1,  0,
                   matrix1);
  MySphereZ(hdc, X1,Y1,Z1,r1, 0,255,255);
  x2 = R2*cos(Beta);
  z2 = R2*sin(Beta);
  SetZRotateMatrix(matrixA, -Alpha);
  SetShiftMatrix(matrixB, -x1,-y1,0);
```

```
  MatrixAxB(matrixB, matrixB, matrixA);
  MatrixAxB(matrix2, matrix1, matrixB);
  MyTransformCoords(&X2,&Y2,&Z2,
                    x2,  0, z2,
                    matrix2);
  MySphereZ(hdc, X2,Y2,Z2,r2, 255,0,0);
  Alpha += stepA;
  Beta  += stepB;
  }
delete []pZbuf;
ReleaseDC(hWnd,hdc);
}
void MySphereZ(HDC hdc,
               double xc,double yc,double zc,int R,
               BYTE red,BYTE gre,BYTE blu)
{
COLORREF clr;
double   R2,r2,z,k;
int      x,y;

R2 = R*R;
for (x=0;x<=R;x++)
  for (y=0;y<=x;y++)
    {
    r2 = x*x + y*y;
    if (r2 > R2) break;
    if (R2 > r2)
       z = sqrt(R2 - r2);
    else z=0;
    k = 1 - r2/R2;
    clr = RGB((BYTE)(k*(double)red),
              (BYTE)(k*(double)gre),
              (BYTE)(k*(double)blu));
    SetPixMyZ(hdc,xc+x,yc+y,zc+z,clr);
    SetPixMyZ(hdc,xc+x,yc-y,zc+z,clr);
    SetPixMyZ(hdc,xc-x,yc+y,zc+z,clr);
    SetPixMyZ(hdc,xc-x,yc-y,zc+z,clr);
    SetPixMyZ(hdc,xc+y,yc+x,zc+z,clr);
    SetPixMyZ(hdc,xc+y,yc-x,zc+z,clr);
    SetPixMyZ(hdc,xc-y,yc+x,zc+z,clr);
    SetPixMyZ(hdc,xc-y,yc-x,zc+z,clr);
    }
}
```

```
oid SetPixMyZ(HDC hdc,int x,int y,float z,COLORREF clr)

ong indx;

f (x<0 || y<0 || x>=horSize || y>=vertSize) return;
ndx = (long)y*horSize + (long)x;
f (z >= pZbuf [indx])
   {
   pZbuf [indx] = z;
   SetPixel(hdc,x,y,clr);
   }
}
```

В приведенных выше примерах вызов функций `SetPixel` — самое критичное место программы с точки зрения быстродействия. Можно усовершенствовать отдельные фрагменты этих примеров, но, как кажется, скорость увеличится не намного. Обычно, чем больше делается вызовов `SetPixel`, тем больше требуется времени на создание изображения. Необходимо искать возможности использовать функции, которые рисуют сразу много пикселов. Это можно считать общим правилом. Однако можно найти и исключения из этого правила — когда процесс расчета точек изображения длится дольше, чем собственно рисование пикселов.

Фрактал Мандельброта

Одним из примеров графики, когда быстродействие функции `SetPixel` не имеет особого значения, можно считать рисование фрактала Мандельброта (рис. 7.7).

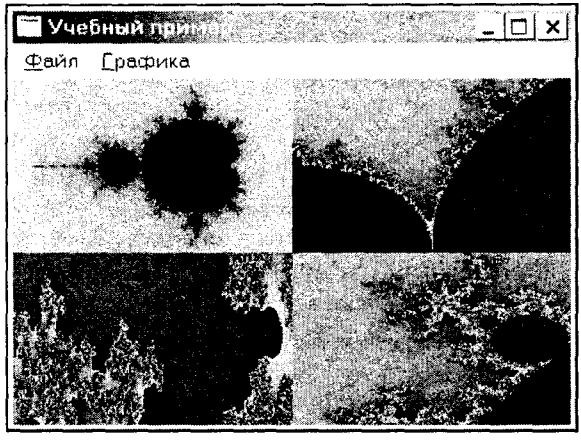


Рис. 7.7. Фрактал Мандельброта и его фрагменты

Рассмотрим текст программы studex10.cpp.

```
//----------(c)Copyright Порев В.Н.-----------
#include "winmain1.cpp"
#include <math.h>
#define MaxIterationsIndex 511
void Mandelbrot(HDC hdc,int xx,int yy,int cx,int cy,
                double minX,double maxX,
                double minY,double maxY);
int Iterations(double x,double y);
COLORREF IndexToColor(int index);

void DrawStudyExample(HWND hWnd)
{
HDC hdc;
RECT rc;

hdc = GetDC(hWnd);
if (hdc == NULL) return;
GetClientRect(hWnd,&rc);
Mandelbrot(hdc, 0, 0,
               rc.right/2, rc.bottom/2,
               -2.2, 1, -1.2, 1.2);
Mandelbrot(hdc, rc.right/2, 0,
           rc.right/2, rc.bottom/2,
           -1, -0.5, -0.5, 0);
Mandelbrot(hdc, rc.right/2,rc.bottom/2,
               rc.right/2,rc.bottom/2,
           -0.75, -0.6, -0.5, -0.35);
Mandelbrot(hdc, 0,rc.bottom/2,
           rc.right/2,rc.bottom/2,
           -0.68, -0.65, -0.37, -0.36);
ReleaseDC(hWnd,hdc);
}
//-аргументы: xx,yy,cx,cy — координаты и размеры окна--
//---------- minX,..,maxY — фрагмент комплексной плоскости
void Mandelbrot(HDC hdc, int xx,int yy,int cx,int cy,
                double minX,double maxX,
                double minY,double maxY)
{
double stepX,stepY;
int i,j,iter;
double x,y;
```

```
tepX = (maxX - minX)/(double)cx;
tepY = (maxY - minY)/(double)cy;
 = minY;
or (j=0;j<cy;j++)
    {
    x = minX;
    for (i=0;i<cx;i++)
        {
        iter = Iterations(x,y);
        SetPixel(hdc,xx+i,yy+j,IndexToColor(iter));
        x += stepX;
        }
    y += stepY;
    }

'//------цикл итераций для одной точки изображения--------
nt Iterations(double x,double y)
{
int i;
double xx,yy,xk,yk;

xx=x;
yy=y;
i=0;
while (xx*xx + yy*yy <= 4.0)
  {
  xk = xx*xx - yy*yy + x;
  yk = 2.0*xx*yy + y;
  xx = xk;
  yy = yk;
  i++;
  if (i >= MaxIterationsIndex) break;
  }
return i;
}
//-----цвет точки в зависимости от числа итераций------
COLORREF IndexToColor(int index)
{
return 8*(MaxIterationsIndex-index);     //эмпирически
}
```

Здесь для каждого пиксела выполняется не больше, чем 512 итераций. Номер последней итерации для каждой точки комплексной плоскости (от 0 до 511)

определяет цвет пиксела в окне отображения. Преобразование номера в цвет сделано в виде отдельной функции `IndexToColor`. Эта функция очень проста, ее следует рассматривать лишь как возможный вариант. Для того чтобы получить более яркие цветные изображения этого же фрактала, можно запрограммировать другие способы преобразования индексов в цвета. Поэкспериментируйте с этим, возможно в результате вы получите такой код функции `IndexToColor`, который будет значительно превышать по объему весь код вычисления фрактала. Кардинально меняет изображение изменение диапазона `(minX, maxX, minY, maxY)`. В нашем примере программы показываются четыре изображения — весь фрактал и три его фрагмента.

Рисование фрагментов фрактала — это не простое увеличение изображения, оно дает все новые и новые детали. Поэкспериментируйте с увеличением фрагментов (интересные изображения располагаются у границы фигуры). Для этого вызовите функцию `Mandelbrot` с соответствующими значениями аргументов `minX,maxX,minY,maxY`.

Трассировка лучей

Еще одним примером программы, для которой быстродействие функции `SetPixel` не критично, может служить реализация метода обратной трассировки лучей.

Здесь использован алгоритм, основанный на локальных преобразованиях координат, в общих чертах уже рассмотренный в главе 4. Мы обсудим несколько усеченный вариант данного алгоритма. Будем выводить только зеркальные и матовые (диффузные) поверхности, заданные плоскими гранями. Тем не менее, это позволяет ощутить основные особенности метода обратной трассировки.

Текст программы `StudeEx13` достаточно объемный.

```
//----------пример трассировки лучей (c) Порев В.Н--------
#include "winmain1.cpp"
#include <math.h>
#define NumGran 8
#define DIFFUSE 0
#define REFLECT 1

struct VERTEX
{
double x,y,z;    //координаты точки в пространстве
};
```

```
truct GRANPARAM    //параметры объектов

nt start;       //индекс первой вершины грани
nt nv;          //число вершин
nt type;        //тип грани
OLORREF clr;    //цвет
};
RANPARAM grandef[] =   //описание свойств граней
{{0, 4,DIFFUSE, RGB(255,224,255)}, //пол
 {4, 4,REFLECT, RGB(0,128,128)},   //3 зеркала
 {8, 4,REFLECT, RGB(0,128,128)},
 {12,4,REFLECT, RGB(0,128,128)},
 {16,3,DIFFUSE, RGB(255,180,0)},   //пирамида
 {19,3,DIFFUSE, RGB(200,180,0)},
 {22,3,DIFFUSE, RGB(200,180,0)},
 {25,3,DIFFUSE, RGB(200,180,0)}};
VERTEX vg[] =            //массив вершин граней
{{-700,200,0},{700,200,0},{700,-500,0},{-700,-500,0},
{-300,-300,0},{300,-300,0},{300,-300,300},{-300,-300,300},
{-300,-300,0},{-300,-300,300},{-500,0,300},{-500,0,0},
{300,-300,0},{300,-300,300},{500,0,300},{500,0,0},
{-80,80,40},{80,80,40},{0,0,200},    //грани пирамиды
{80,80,40},{80,-80,40},{0,0,200},
{-80,80,40},{-80,-80,40},{0,0,200},
{-80,-80,40},{80,-80,40},{0,0,200}};
VERTEX camerapos = {250,  500, 200},   //положение камеры
       cameradir = {-1.3, -5, -1},
       lightpos  = {-100,250,350};     //источник света
VERTEX pgn[4];             //служебный массив для вершин

void CreateStartRay(double *M,double *MV,int x,int y);
COLORREF Ray(double *M,int ngran,int level);
COLORREF DiffuseRay(double *M,int ng);
BOOL PointInShadow(double *M,int ng);
BOOL NearestGranPoint(double *M,double *zmin,
                      int *ng,int no);
BOOL IntersectPoint(double *zp,int nv);
void MatrixAxB(double *dest,double *A, double *B);
void SetKoordTransform(double *M,VERTEX *rv,
                       double x,double y,double z);
```

```
void TransformKoord(double *M,VERTEX *dest,VERTEX *src);
void NormalToGran(double *M,VERTEX *normal,int ng);
void NormalVector(VERTEX*nv,VERTEX v1,VERTEX v2,VERTEX v3);
void Normalize(VERTEX *v);
void ReflectionVector(VERTEX *vr,VERTEX *normal);

void DrawStudyExample(HWND hWnd)
{
HDC hdc;
RECT rc;
int x,y;
double M[16],MV[16];    //матрицы преобразований
COLORREF c;

hdc = GetDC(hWnd);
if (hdc == NULL) return;
SetKoordTransform(MV,&cameradir,       //видовая матрица
                      camerapos.x,     //будет учитывать
                      camerapos.y,     //положение камеры
                      camerapos.z);
GetClientRect(hWnd,&rc);
for (y=-rc.bottom/2; y<rc.bottom/2; y++)     //основной
  for (x=-rc.right/2; x<rc.right/2; x++)     //цикл
    {
    CreateStartRay(M,MV,x,y);
    c = Ray(M, -1, 0);          //первичный луч
    SetPixel(hdc,x+rc.right/2,
                 y+rc.bottom/2, c);
    }
ReleaseDC(hWnd,hdc);
}
//-------формируем матрицу для луча из камеры-------
void CreateStartRay(double *M,double *MV,int x,int y)
{
VERTEX dir;

dir.x = x;
dir.y = y;
dir.z = -250;    //расстояние до плоскости проецирования
SetKoordTransform(M, &dir, x,y,-250);
```

```
trixAxB(M, M, MV); //учитываем видовое преобразование

--------основная процедура трассировки луча--------
LORREF Ray(double *M,int ngran,int level)

it ng;
ouble z;
RTEX normal,dir;
ouble Md[16],Ms[16];
OLORREF clr;
nt r,g,b,r1,g1,b1;

f (level > 20) return 0;    //защита от зацикливания
f (NearestGranPoint(M,&z,&ng,ngran))      //найдена точка
  {                                       //пересечения луча
  NormalToGran(M,&normal,ng);             //нормаль к грани
  r = GetRValue(grandef[ng].clr);
  g = GetGValue(grandef[ng].clr);
  b = GetBValue(grandef[ng].clr);
  if (grandef[ng].type == DIFFUSE)        //матовая грань
     {
     SetKoordTransform(Md,&normal,0,0,z);
     MatrixAxB(Md, Md, M);
     return DiffuseRay(Md,ng);
     }
  if (grandef[ng].type == REFLECT)            //зеркало
     {
     ReflectionVector(&dir, &normal);
     SetKoordTransform(Ms,&dir,0,0,z);
     MatrixAxB(Ms, Ms, M);
     clr = Ray(Ms, ng, level+1);     //отраженный луч
     r1 = GetRValue(clr);
     g1 = GetGValue(clr);
     b1 = GetBValue(clr);
     return RGB(r/10 + (double)r1*0.9,
                g/10 + (double)g1*0.9,
                b/10 + (double)b1*0.9);
     }
  return grandef[ng].clr;
  }
```

```
else return RGB(255,255,255);
}

COLORREF DiffuseRay(double *M,int ng)
{
VERTEX v;
int r,g,b;
double Kd;
double Ml[16];

r = GetRValue(grandef[ng].clr);     //цвет грани
g = GetGValue(grandef[ng].clr);
b = GetBValue(grandef[ng].clr);
TransformKoord(M,&v, &lightpos);
Normalize(&v);        //вектор на источник света
SetKoordTransform(Ml,&v,0,0,0);
MatrixAxB(Ml, Ml, M);
if (PointInShadow(Ml,ng))       //для точки в тени
   return RGB(r/3, g/3, b/3);  //уменьшаем яркость
Kd = -v.z;
if (Kd < 0.33) Kd=0.33;         //фоновая подсветка
if (Kd > 1) Kd=1;
return RGB((double)r*Kd,
           (double)g*Kd,
           (double)b*Kd);
}
//------направляем луч на источник света---------
BOOL PointInShadow(double *M,int ng)
{
int i,j;
double z;
VERTEX v;

TransformKoord(M, &v, &lightpos);
for (i=0;i<NumGran;i++)
  {
  if (i == ng) continue;
  for (j=0; j<grandef[i].nv; j++)
     TransformKoord(M, &pgn[j], &vg[grandef[i].start+j]);
  if (IntersectPoint(&z, grandef[i].nv))
```

```
    if ((z < 0)&&(z > v.z))
       return TRUE;     //источник света закрыт гранью
  }
:eturn FALSE;       //нет препятствий на пути луча света

'/------точка пересечения луча с ближайшей гранью
3OOL NearestGranPoint(double *M,double *zmin,
                      int *ng,int no)
{
int i,j,n=-1;
double z,zm;
BOOL first=TRUE;

for (i=0;i<NumGran;i++)
  {
  if (i == no) continue;  //из этой грани исходит луч
  for (j=0; j<grandef[i].nv; j++)
     TransformKoord(M, &pgn[j], &vg[grandef[i].start + j]);
  if (IntersectPoint(&z, grandef[i].nv))
     if (z < 0)
        {
        if (first)
           {
           n=i; zm = z; first=FALSE;
           }
        else
           {
           if (zm < z)
              {
              zm = z; n=i;
              }
           }
        }
  }
if (n >= 0)         //грань найдена
   {
   *zmin = zm;      //координата точки пересечения
   *ng = n;         //номер ближайшей (видимой) грани
   return TRUE;
   }
```

```
return FALSE;
}
//определение координаты Z точки пересечения с гранью
//---вершины грани должны быть в массиве pgn[]-----
BOOL IntersectPoint(double *zp,int nv)
{
int i,nhor,next,xmin,xmax,ymin,ymax;
double x[4],z[4];
double y1,y2;

xmin=xmax=pgn[0].x;
ymin=ymax=pgn[0].y;
for (i=1;i<nv;i++)
  {
  if (xmin > pgn[i].x) xmin = pgn[i].x;
  if (ymin > pgn[i].y) ymin = pgn[i].y;
  if (xmax < pgn[i].x) xmax = pgn[i].x;
  if (ymax < pgn[i].y) ymax = pgn[i].y;
  }
if ((xmin > 0)||(xmax < 0)||(ymin > 0)||(ymax < 0))
   return FALSE;
nhor = 0;
for (i=0; i<nv; i++)
  {
  next = i+1;
  if (next >= nv) next=0;
  y1 = pgn[i].y;
  y2 = pgn[next].y;
  if ((0 >= y1)&&(0 < y2)||(0 <= y1)&&(0 > y2))
     {
     x[nhor]=pgn[i].x-(pgn[next].x-pgn[i].x)*y1/(y2-y1);
     z[nhor]=pgn[i].z-(pgn[next].z-pgn[i].z)*y1/(y2-y1);
     nhor++;
     }
  }
if (nhor != 2) return FALSE;
if (x[1]==x[0])
   {
   if (x[1] == 0)
      {
      *zp = 0.5*(z[0]+z[1]);
```

```
      return TRUE;
      }
   return FALSE;
   }
if (((x[0] <= 0)&&(x[1] >= 0))||
    ((x[0] >= 0)&&(x[1] <= 0)))
   {
   *zp = z[0]-x[0]*(z[1]-z[0])/(x[1]-x[0]);
   return TRUE;
   }
return FALSE;
}

//--------------перемножение матриц--------------------
void MatrixAxB(double *dest,double *A, double *B)
{
int indx,i,j,k;
double tmp[12];    //последняя строка не нужна

for (indx=0; indx<12; indx++)
    {
    i = indx & 0xfc;      //i кратно 4
    j = indx & 3;         //j = 0,1,2,3
    tmp[indx]=0;
    for (k=0; k<4; k++)
        {
        tmp[indx] += A[i]*B[j];
        i++;
        j += 4;
        }
    }
memcpy(dest,tmp,12*sizeof(double));
dest[12]=dest[13]=dest[14]=0; dest[15]=1;
}

//вектор (-rv) указывает новое направление оси Z
//координаты сдвига (x,y,z) лучше задавать раздельно
void SetKoordTransform(double *M,VERTEX *rv,
                       double x,double y,double z)
{
double R,r;
double csA,snA,csB,snB;
```

```
for (int i=0;i<16;i++) M[i]=0;
M[0]=M[5]=M[10]=M[15]=1;          //единичная матрица
if ((rv->x == 0)&&(rv->y == 0))   //нужен только сдвиг
   if (rv->z >= 0)
      {
      M[3]=-x; M[7]=-y; M[11]=-z;
      return;
      }
   else
      {
      M[3]=x; M[7]=-y; M[11]=z;
      return;
      }
R = sqrt(rv->x*rv->x + rv->y*rv->y + rv->z*rv->z);
r = sqrt(rv->x*rv->x + rv->y*rv->y);
csB = -rv->z/R;
snB = r/R;
csA = -rv->y/r;
snA = -rv->x/r;
M[0] = csA;
M[1] = -snA;
M[2] = 0;
M[3] = -M[0]*x - M[1]*y;
M[4] = snA*csB;
M[5] = csA*csB;
M[6] = -snB;
M[7] = -M[4]*x - M[5]*y - M[6]*z;
M[8] = snA*snB;
M[9] = csA*snB;
M[10] = csB;
M[11] = -M[8]*x - M[9]*y - M[10]*z;
}

void TransformKoord(double *M,VERTEX *dest,VERTEX *src)
{
VERTEX tmp;

tmp.x = M[0]*src->x + M[1]*src->y +  M[2]*src->z + M[3];
tmp.y = M[4]*src->x + M[5]*src->y +  M[6]*src->z + M[7];
tmp.z = M[8]*src->x + M[9]*src->y + M[10]*src->z + M[11];
*dest = tmp;
}
```

```
/-----вычисление единичного вектора нормали к грани------
/--------ng - номер грани в общем списке-----------------
/---M[ ] - матрица преобразования локальных координат ---
oid NormalToGran(double *M,VERTEX *normal,int ng)

ERTEX v1,v2,v3;

'ransformKoord(M,&v1, &vg[grandef[ng].start]);
'ransformKoord(M,&v2, &vg[grandef[ng].start+1]);
'ransformKoord(M,&v3, &vg[grandef[ng].start+2]);
Jormalvector(normal,v1,v2,v3);
Jormalize(normal);
}

//---------нормаль к треугольнику (v1, v2, v3)----------
void NormalVector(VERTEX *nv,VERTEX v1,VERTEX v2,VERTEX v3)
{
v2.x -= v1.x;
v2.y -= v1.y;
v2.z -= v1.z;
v3.x -= v1.x;
v3.y -= v1.y;
v3.z -= v1.z;
nv->z =  v2.x * v3.y - v3.x * v2.y;
nv->x =  v2.y * v3.z - v3.y * v2.z;
nv->y = -v2.x * v3.z + v3.x * v2.z;
if (nv->z < 0)    //выбираем нормаль видимой стороны
   {
   nv->x = -nv->x;
   nv->y = -nv->y;
   nv->z = -nv->z;
   }
}

//----------делаем вектор единичной длины-------
void Normalize(VERTEX *v)
{
double R;

R = sqrt(v->x*v->x + v->y*v->y + v->z*v->z);
v->x /= R;
```

```
v->y /= R;
v->z /= R;
}

//--------вектор луча зеркального отражения-----------
//------падающий луч здесь всегда вдоль оси Z----------
void ReflectionVector(VERTEX *vr,VERTEX *n)
{
vr->x = n->x*n->z;
vr->y = n->y*n->z;
vr->z = n->z*n->z - 0.5;
}
```

Результат работы программы `StudEx13` изображен на рис. 7.8.

Рис. 7.8. Пирамида и три зеркала

В этом примере объекты являются либо только зеркальными, либо только диффузными. Кроме того, не показывается сам источник света — предполагается, что он не виден ни прямо, ни в отраженном виде. Источник света здесь один — точечный, светит равномерно во все стороны белым светом.

Поскольку объектов немного и они просты, то здесь не используется метод оболочек.

Данную программу можно усовершенствовать для показа более сложных изображений. Например, предусмотреть несколько разноцветных источников

:вета, зеркальные блики, прозрачность объектов. Сделайте это самостоятель-ю в качестве упражнения.

7.2. Линии

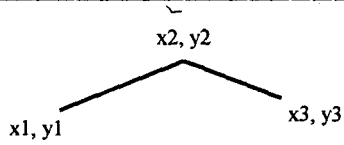
В состав API Windows входит несколько функций, которые рисуют прямые и кривые линии:

- **AngleArc** — дуга окружности с заданием углов;
- **Arc, ArcTo** — дуга эллипса;
- **LineDDA** — вычисление координат пикселов отрезка прямой линии и вызов определяемой пользователем функции вывода пикселов;
- **LineTo** — рисование отрезка прямой линии от текущей позиции к заданной точке;
- **MoveToEx** — задание текущей позиции графического вывода;
- **PolyBezier, PolyBezierTo** — один или несколько связанных между собой кубических сплайнов Безье;
- **PolyDraw** — несколько связанных отрезков прямых и сплайнов Безье;
- **Polyline, PolylineTo** — ломаная линия из многих связанных между собой отрезков прямых (полилиния);
- **PolyPolyline** — несколько полилиний как единый объект.

Примеры для линий — графических примитивов API Windows — даны в табл. 7.1.

Таблица 7.1

Что рисуется	Программный код

	`MoveToEx(hdc,x1,y1,NULL);`
	`LineTo(hdc,x2,y2);`
	`LineTo(hdc,x3,y3);`

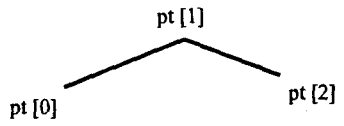
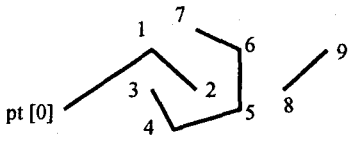
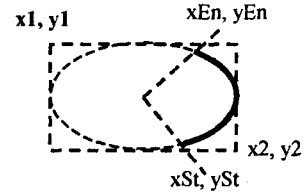
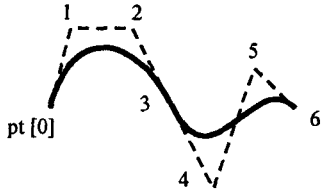
	`POINT pt[3];`
	
	`Polyline(hdc, pt, 3);`

Таблица 7.1 *(окончание)*

Что рисуется	Программный код

	`POINT pt[10];`
	`DWORD npt[3]={3,5,2};`
	
	`PolyPolyline(hdc, pt,`
	`npt, 3);`

	`Arc(hdc, x1, y1,`
	`x2, y2,`
	`xSt, ySt,`
	`xEn, yEn);`

	`POINT pt[7];`
	
	`PolyBezier(hdc,pt,7);`

Стиль линии. Перо

В терминологии Windows API *перо* описывает следующие характеристики линии — цвет, толщину и стиль (пунктир). Перо — один из атрибутов контекста графического устройства. По умолчанию в контекст выбрано перо, которое соответствует черной тонкой непрерывной линии. Такое перо относится к *стандартным перьям*.

Все линии, рисуемые вашей программой с помощью функций GDI Windows API, выводятся одним и тем же цветом, имеют одинаковую толщину и тот же стиль до тех пор, пока не изменить перо. Функции рисования линий не имеют аргументов для указания текущих атрибутов линий (это характерно для графических библиотек, в которых подобные характеристики рисуемых объектов записываются как глобальные переменные, чтобы уменьшить количество аргументов вызова функций рисования).

Перо относится ко всем линиям и контурам фигур.

Чтобы начать рисовать линии, например, другим цветом, необходимо создать новое перо и выбрать его в контексте графического устройства. Отныне все

инии будут рисоваться данным пером до тех пор, пока вы не выберете в контексте устройства очередное перо.

Важно то, что, создавая новое перо, нужно обязательно позаботиться об уничтожении предыдущего пера. Перо есть *объект* GDI, для него выделяется специальная область памяти. Перо существует до тех пор, пока его не уничтожить. Завершение работы прикладной программы может и не привести к автоматическому уничтожению объектов GDI и освобождению памяти компьютера. Своевременное уничтожение неиспользуемых объектов GDI возлагается на вашу программу. Иначе для некоторых версий Windows могут возникнуть утечки памяти, накопление которых может привести к нежелательным последствиям.

Не следует пытаться уничтожить стандартные перья.

Для определения стиля линий используются функции, которые имеют в своем названии ---Pen---. Типичная последовательность для вывода линии с заданным стилем может быть такой:

```
HPEN hpen, hPenOld;
. . . .
hPen = CreatePen(PS_DASHDOT, 1, RGB(255,0,0));
hPenOld = (HPEN)SelectObject(hdc,hPen);

. . . .        // здесь рисуем линии

SelectObject(hdc,hPenOld); //выбираем предыдущее перо
DeleteObject(hPen);        //уничтожаем наше перо
```

Здесь создается перо, которое соответствует тонкой красной штрихпунктирной линии. После использования перо уничтожается.

Меридианы и параллели

Рассмотрим пример программы, в которой используются функции `MoveToEx` и `LineTo` для рисования меридианов и параллелей шара.

Текст программы `studex20.cpp`:

```
//------------------(c)Copyright Порев В.Н.----------------
#include "winmain1.cpp"
#include <math.h>

void DrawMyEarth(HDC hdc,double alpha,double beta,
                          int cxw,int cyw);
```

```
void MeridOrParal(HDC hdc,int cxw,int cyw,double R,
                  double alpha,double beta,BOOL mer);
void RotateXYZ(double *X,double *Y,double *Z,
               double x,double y,double z,
               double a,double b);

void DrawStudyExample(HWND hWnd)
{
HDC hdc,hdcWin;
RECT rc;
HBITMAP hBitmap;
HCURSOR hCurOld;

hCurOld = SetCursor(LoadCursor(NULL,IDC_WAIT));
GetClientRect(hWnd,&rc);
hdcWin = GetDC(hWnd);
hdc = CreateCompatibleDC(hdcWin);
hBitmap = CreateCompatibleBitmap(hdcWin,
                                 rc.right, rc.bottom);
SelectObject(hdc,hBitmap);
PatBlt(hdc,0,0,rc. right, rc. bottom,WHITENESS);
for (int alpha=0;alpha<=180;alpha++)      //цикл анимации
    {
    DrawMyEarth(hdc,alpha,45,rc.right, rc.bottom);
    BitBlt(hdcWin,0,0, rc.right, rc.bottom,
           hdc,0,0,SRCCOPY);
    }
DeleteDC(hdc);
DeleteObject(hBitmap);
ReleaseDC(hWnd,hdcWin);
SetCursor(hCurOld);       //восстанавливаем форму курсора
}

void DrawMyEarth(HDC hdc,double alpha,double beta,
                 int cxw,int cyw)
{
int R;

if (cxw <= cyw)
     R = cxw/2-10;
else R = cyw/2-10;
Ellipse(hdc,cxw/2-R,cyw/2-R,cxw/2+R,cyw/2+R);
```

```
eridOrParal(hdc,cxw,cyw,R,alpha,beta,TRUE);
eridOrParal(hdc,cxw,cyw,R,alpha,beta,FALSE);


oid MeridOrParal(HDC hdc,int cxw,int cyw,double R,
                 double alpha,double beta,BOOL mer)

nt i,j,start_i,end_i,start_j,end_j;
OOL start;
louble x,y,z,X,Y,Z,B,L,rad;
louble *ai,*aj;

ad = M_PI/180.0;
lpha *= rad;            //из градусов в радианы
eta  *= rad;
f (mer)                 //рисование меридианов
   {
   start_i = 0;
   end_i = 360;
   start_j = -90;
   end_j   = 90;
   ai = &L;             //долгота
   aj = &B;             //широта
   }
lse                     //рисование параллелей
   {
   start_i = -80;
   end_i = 80;
   start_j = 0;
   end_j = 360;
   ai = &B;
   aj = &L;
   }
or (i=start_i; i<end_i; i += 10)
   {
    start=FALSE;
    *ai = (double)i * rad;
    for (j=start_j; j<=end_j; j += 10)
        {
        *aj = (double)j * rad;
//------координаты точки поверхности шара--------
        x = R*cos(B)*sin(L);
        y = R*cos(B)*cos(L);
        z = R*sin(B);
        RotateXYZ(&X,&Y,&Z, x,y,z, alpha,beta);
```

```
        if (start)
           {
           if (Z < 0) start=FALSE;
           else LineTo(hdc,X+cxw/2, Y+cyw/2);
           }
        else
           {
           if (Z >= 0)          //уже на видимом полушарии
              {
              start=TRUE;
              MoveToEx(hdc,X+cxw/2, Y+cyw/2,NULL);
              }
           }
        }
    }
}

//------------ поворот осей на углы a и b--------------
void RotateXYZ(double *X,double *Y,double *Z,
               double x,double y,double z,
               double a,double b)
{
*X = x*cos(a) - y*sin(a);
*Y = x*sin(a)*cos(b) + y*cos(a)*cos(b) - z*sin(b);
*Z = x*sin(a)*sin(b) + y*cos(a)*sin(b) + z*cos(b);
}
```

Изображение шара приведено на рис. 7.9.

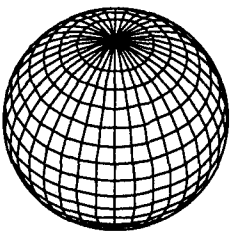


Рис. 7.9. Меридианы и параллели шара

Запустите программу, выберите пункт меню "Графика" и вы увидите шар, который, как кажется, вращается вокруг оси. Здесь вращается не шар, а система координат. Для этого использовано соответствующее преобразование координат при создании проекции — повороты координат. Углы наклона камеры в этом примере следующие: α=0...180, β=45 градусов. Меридианы и параллели рисуются только для тех точек поверхности, для которых координата Z больше нуля (в повернутой системе). Положительные значения Z соответствуют видимой части шара. Рисование меридианов и параллелей сделано в виде одной функции `MeridOrParal` для уменьшения размера текста программы. Необходимо отметить, что эта программа разработана как раз для шара. Чтобы

нарисовать подобным образом, например, эллипсоид, необходимо сделать существенные изменения в тексте программы.

Фрактал из линий

В следующем примере программы мы используем линии для рисования фрактала. Этот фрактал немного напоминает папоротник (рис. 7.10). Используем перья различных оттенков зеленого цвета для имитации ствола, ветвей и листвы.

В этом примере рисуются линии различной толщины: для листвы линии толстые, а для ствола и ветвей — тонкие (можно и наоборот — для листвы более толстые линии).

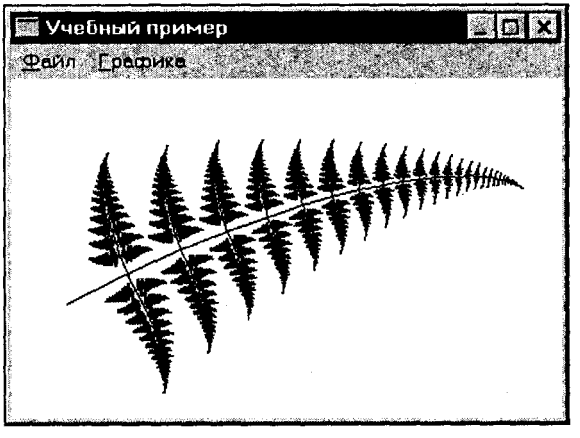


Рис. 7.10. Фрактал из линий

Текст программы `studex21.cpp`:

```
//-------(c)Copyright Порев В.Н.----------
#include "winmain1.cpp"
#include <math.h>
  }
HDC hdc;
double A,B,C,D,E,F,G,H,K,L;
int MaxIter = 5;
int ColorStep=0;

void SetKoef(double alpha, double beta,
             double k, double k1);
void OneIteration(double x1,double y1,
                  double x2,double y2,int num);
```

```
void DrawStudyExample(HWND hWnd)
{
RECT rc;

hdc = GetDC(hWnd);
GetClientRect(hWnd,&rc);
ColorStep = 255/MaxIter;
SetKoef(2, 86, 0.14, 0.31);
OneIteration(rc.right/10, 2*rc.bottom/3,
             7*rc.right/8, 0, 0);
ReleaseDC(hWnd,hdc);
}

void OneIteration(double x1,double y1,
                  double x2,double y2,int num)
{
double x3,y3,x4,y4,x5,y5,x6,y6,x7,y7;
HPEN hPenOld,hPen;

if (num > MaxIter) return;
if ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) < 0.7) return;

x3 = (x2-x1)*A - (y2-y1)*B + x1;
y3 = (x2-x1)*B + (y2-y1)*A + y1;
x4 = x1*C + x3*D;
y4 = y1*C + y3*D;
x5 = x4*E + x3*F;
y5 = y4*E + y3*F;
x6 = (x5 -x4)*G - (y5 - y4)*H + x4;
y6 = (x5 -x4)*H + (y5 - y4)*G + y4;
x7 = (x5 -x4)*G + (y5 - y4)*H + x4;
y7 = -(x5 -x4)*H + (y5 - y4)*G + y4;

//---------рисуем линию ветви сплошной линией------
//толщина линии и цвет согласно номеру цикла итераций
//-----------цвет линии - градации зеленого---------
hPen = CreatePen(PS_SOLID,
                 (2*num)/3,
                  RGB(0,ColorStep*num,0));
hPenOld = (HPEN)SelectObject(hdc, hPen);
MoveToEx(hdc,x1,y1,NULL);
LineTo(hdc,x4,y4);
SelectObject(hdc, hPenOld);
DeleteObject(hPen);
```

```
//----------продолжаем итерации-------------------
OneIteration(x4,y4,x6,y6,num+1); //сначала ветви
OneIteration(x4,y4,x7,y7,num+1);
OneIteration(x4,y4,x3,y3,num);   //ствол
}

void SetKoef(double alpha,double beta,
             double k,double k1)
{
alpha *= M_PI/180;
beta *= M_PI/180;
A = cos(alpha);
B = sin(alpha);
C = 1-k;
D = k;
E = 1-k1;
F = k1;
G = cos(beta);
H = sin(beta);
}
```

7.3. Фигуры

В API Windows есть несколько графических примитивов, которые предназначены для рисования фигур с заполнением:

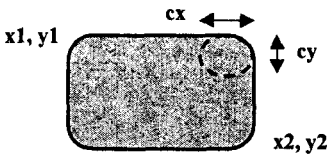
- **Chord** — хорда эллипса;
- **Ellipse** — эллипс;
- **Pie** — сектор эллипса;
- **Polygon** — полигон;
- **PolyPolygon** — несколько полигонов и (или) полигоны с пустотами;
- **Rectangle** — прямоугольник;
- **RoundRect** — прямоугольник со скругленными углами.

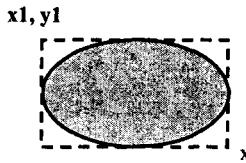
Таблица 7.2

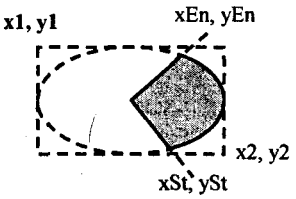
Что рисуется	Программный код
x1, y1  x2, y2	`Rectangle(hdc,x1,y1,` `x2,y2);`

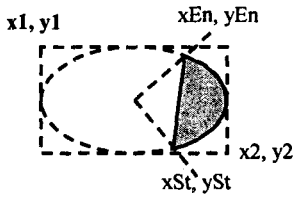
Таблица 7.2 *(окончание)*

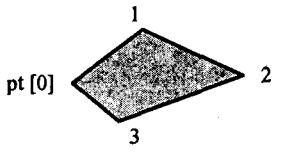
Что рисуется	Программный код

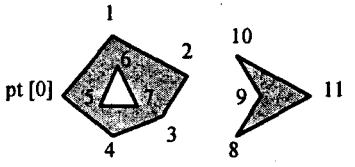
	`RoundRect(hdc,x1,y1,`
	`x2,y2,`
	`cx,cy);`

	`Ellipse(hdc, x1, y1,` `x2, y2);`

	`Pie(hdc, x1, y1,`
	`x2, y2,`
	`xSt, ySt,`
	`xEn, yEn);`

	`Chord(hdc, x1, y1,`
	`x2, y2,`
	`xSt, ySt,`
	`xEn, yEn);`

	`POINT pt[4];`
	
	`Polygon(hdc, pt, 4);`

	`POINT pt[12];`
	`int npt[3] = {5,3,4};`
	
	`PolyPolygon(hdc, pt,`
	`npt, 3);`

Стиль заполнения. Кисть

По умолчанию в контексте графического устройства устанавливается стиль заполнения сплошным белым цветом. Для того чтобы рисовать определенную фигуру другим стилем, необходимо создать соответствующую кисть. Кисть и стиль заполнения — синонимы в API Windows.

Кисть — это объект GDI. Он требует памяти. Кроме того, все кисти, созданные во время работы программы, необходимо уничтожить, иначе они могут остаться в памяти после завершения программы. Общая схема использования кистей такая же, как и для перьев:

1. Создание кисти, выбор ее в контекст.
2. Рисование фигур с заполнением.
3. Освобождение контекста, уничтожение кисти.

Сплошная кисть создается функцией `CreateSolidBrush`. Рассмотрим пример использования оранжевой кисти.

```
HBRUSH hBrush, hBrushOld;

hBrush = CreateSolidBrush(RGB(255,128,0));
hBrushOld = (HBRUSH)SelectObject(hdc,hBrush);
  . . . .       //здесь рисуем фигуры
  . . . .
SelectObject(hdc,hBrushOld);
DeleteObject(hBrush);
```

Штриховая кисть создается функцией `CreateHatchBrash`.

```
CreateHatchBrush(HS_DIAGCROSS, RGB(0,0,255));
```

Кисть с заданием растрового шаблона — `CreatePatternBrush`.

```
HBITMAP hBitmap;
HBRUSH hBrush;
WORD wBits[] = {0xFF,0x0C,0x0C,0x0C,0xFF,0xC0,0xC0,0xC0};

hBitmap = CreateBitmap(8,8,1,1,(LPVOID)WBits);
hBrush  = CreatePatternBrush(hBitmap);
```

Рисование поверхности

Рассмотрим пример программы для рисования поверхности, заданной в виде функции $z = f(x, y)$, например:

$$z = \frac{\cos\left(b\sqrt{x^2 + y^2}\right)}{1 + c\sqrt{x^2 + y^2}}.$$

Для рисования с удалением невидимых точек поверхности используем метод сортировки граней по глубине, а точнее, будем прямо рисовать грани от самых дальних к самым близким. Каждую грань можно рисовать четырехугольником-полигоном (рис. 7.11.).

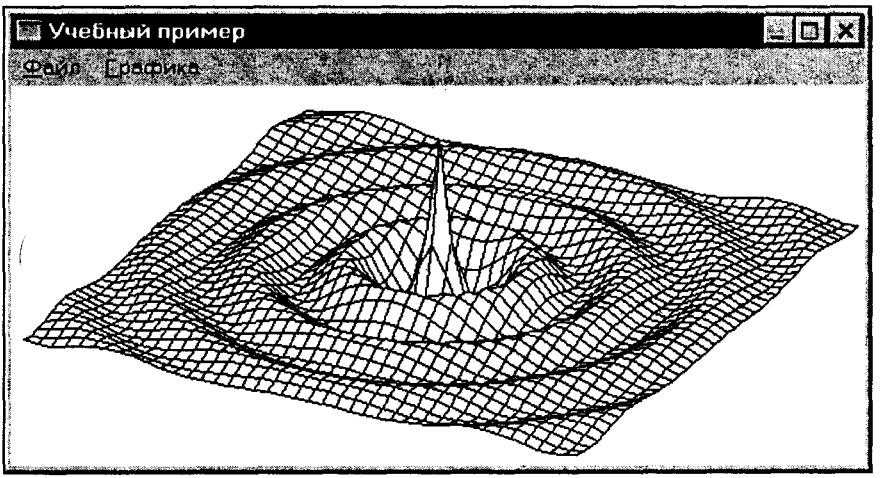


Рис. 7.11. Поверхность $z = f(x, y)$

```
//------график функции z=f(x,y)---(c) Порев В.Н.------
#include "winmain1.cpp"
#include <math.h>

double Aview,Bview,Cview,Dview,Eview,
       Fview,Gview,Hview,Xview,Yview;
double kxView = 10,kyView = 10,kzView = 100;
double kxFunc=0.1,kyFunc=0.1,dxFunc=0,dyFunc=0;

void DrawSurface(HDC hdc,int nx,int ny,int dx,int dy);
double MyFunction(double x,double y);
double SurfaceKoordX(double x);
double SurfaceKoordY(double y);
```

```
void InitViewTranform(double alpha, double beta,
                      int cx, int cy);
void ScreenKoord(POINT *pt,double x,double y,double z);

void DrawStudyExample(HWND hWnd)
{
RECT rc;
HDC hdc;

hdc = GetDC(hWnd);
GetClientRect(hWnd,&rc);
InitViewTranform(27, 65, rc.right, rc.bottom);
PatBlt(hdc,0, 0, rc.right, rc.bottom, WHITENESS);
DrawSurface(hdc,200,200,10,10);
ReleaseDC(hWnd,hdc);
}

void DrawSurface(HDC hdc,int nx,int ny,int dx,int dy)
{
int indx,indy;
double x,y,z;
POINT pt[4];

for (indy=-ny;indy<=ny;indy+=dy)
    for (indx=-nx;indx<=nx;indx+=dx)
        {
        x = SurfaceKoordX(indx);
        y = SurfaceKoordY(indy);
        z = MyFunction(x,y);
        ScreenKoord(&pt[0], x,y,z);
        x = SurfaceKoordX(indx+dx);
        y = SurfaceKoordY(indy);
        z = MyFunction(x,y);
        ScreenKoord(&pt[1], x,y,z);
        x = SurfaceKoordX(indx+dx);
        y = SurfaceKoordY(indy+dy);
        z = MyFunction(x,y);
        ScreenKoord(&pt[2], x,y,z);
        x = SurfaceKoordX(indx);
        y = SurfaceKoordY(indy+dy);
        z = MyFunction(x,y);
        ScreenKoord(&pt[3], x,y,z);
        Polygon(hdc,pt,4);
        }
}
```

```
//-------------------------z = f(x,y)----------------------
double MyFunction(double x,double y)
{
double r,b=1,c=1;

r = sqrt(x*x + y*y);
return cos(r*b)/(r*c+1);
}

//--пересчет индексов узлов в координаты поверхности -
double SurfaceKoordX(double x)
{
return x * kxFunc + dxFunc;
}

double SurfaceKoordY(double y)
{
return y * kyFunc + dyFunc;
}
//--------определение коэффициентов проекции показа--------
void InitViewTranform(double alpha, double beta,
                      int cx, int cy)
{
alpha *= M_PI/180.0;
beta *= M_PI/180.0;
Aview = cos(alpha);
Bview = -sin(alpha);
Cview = sin(alpha)*cos(beta);
Dview = cos(alpha)*cos(beta);
Eview = -sin(beta);
Fview = sin(alpha)*sin(beta);
Gview = cos(alpha)*sin(beta);
Hview = cos(beta);
Xview = cx/2;
Yview = cy/2;
}
//--------------вычисление экранных координат-------------
void ScreenKoord(POINT *pt,double x,double y,double z)
{
x *= kxView;
y *= kyView;
z *= kzView;
pt->x = x*Aview + y*Bview + Xview;
pt->y = x*Cview + y*Dview + z*Eview + Yview;
}
```

7.4. Шрифт TrueType

Рассмотрим пример использования шрифтов TrueType (рис. 7.12).

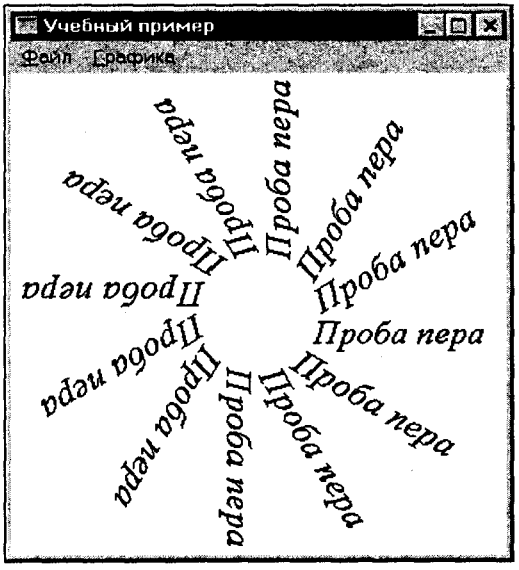


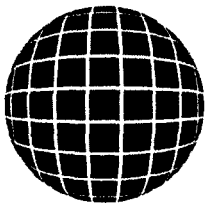
Рис. 7.12. Использование шрифта TrueType

Текст программы `studex23.cpp`:

```
//-------(c)Copyright Порев В.Н.----------
#include "winmain1.cpp"
#include <math.h>
void DrawMyString(HDC hdc,int x, int y,
                        int a, int size, char *s);
void DrawStudyExample(HWND hWnd)
{
HDC hdc;
RECT rc;
int bk;

GetClientRect(hWnd,&rc);
hdc = GetDC(hWnd);
bk = SetBkMode(hdc,TRANSPARENT);
for (int a=0; a<3600; a += 300)
    DrawMyString(hdc,rc.right/2, rc.bottom/2,
                     a, 30, "      Проба пера");
```

```
SetBkMode(hdc,bk);
ReleaseDC(hWnd,hdc);
}

void DrawMyString(HDC hdc,int x, int y,
                          int a, int size, char *s)
{
LOGFONT lf;
HFONT hFont,hFontOld;

memset(&lf,0,sizeof(LOGFONT));
lf.lfHeight = size;
strcpy(lf.lfFaceName,"Times New Roman Cyr");
lf.lfEscapement = a;              //угол наклона
lf.lfWeight = FW_NORMAL;
lf.lfItalic = 1;
lf.lfCharSet = DEFAULT_CHARSET;
lf.lfOutPrecision = OUT_TT_PRECIS;
hFont = CreateFontIndirect(&lf);
if (hFont)
   {
   hFontOld = (HFONT)SelectObject(hdc,hFont);
   TextOut(hdc,x,y,s,strlen(s));
   SelectObject(hdc,hFontOld);
   DeleteObject(hFont);
   }
}
```

В некоторых версиях Windows, возможно, эта программа не будет корректно работать. Может потребоваться задать другое имя шрифта. Но для того, чтобы буквы показывались с наклоном, этот шрифт обязательно должен быть типа TrueType (подойдет и OpenType для Win2000).



ГЛАВА 8

Примеры использования классов языка C++

Рассмотрим пример графической программы, создающей изображения объектов на основе нескольких простых элементов (рис. 8.1).

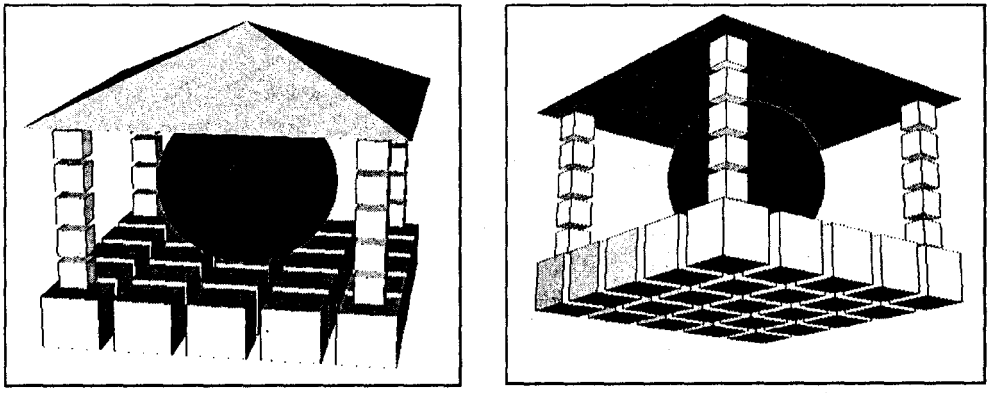


Рис. 8.1. Трехмерные объекты в различных ракурсах показа

Используем *объектно-ориентированную* методологию. Каждый элемент будем считать объектом трехмерного пространства, а несколько таких объектов образовывают модель сложного объекта. Для описания объектов используем классы C++. Процитируем автора языка C++ *Б. Страуструпа:* "Определите, какие классы вам нужны; предусмотрите полный набор операций для каждого класса; опишите общие черты явным образом, используя наследование" [24].

Сложный пространственный объект в нашей программе построим с использованием таких элементов: куб, сфера и пирамида. Фундамент и колонны бу-

дем считать производными элементами, их определим как множество кубов, специальным образом располагающихся в пространстве (рис. 8.2).

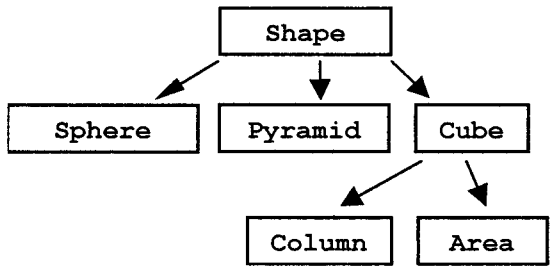


Рис. 8.2. Иерархия классов

В качестве базового элемента определим абстрактный класс фигуры с такими свойствами: размер, цвет, расположение в пространстве, описываемое координатами ее центра. Также предусмотрим для фигур возможность перемещения, изменения размера, цвета и возможность быть нарисованной. Такие общие свойства выразим в классе `Shape`. В этом классе также предусмотрим операцию преобразования координат для отображения в определенной проекции (функция-член `PrepareVertex`).

Теперь обсудим способ отображения объектов. Поскольку у нас есть элементы-многогранники (куб и пирамида), то можно было бы использовать достаточно быстродействующую функцию Polygon API Windows для рисования граней. А удаление невидимых точек осуществлять сортировкой граней по глубине. Однако такой способ отображения в нашем случае не приемлем. Чуть позже мы покажем почему, а пока что обсудим довольно интересные нюансы объектно-ориентированного стиля программирования. Если бы у нас все объекты были многогранниками, то сортировка граней по глубине означала бы определенную последовательность рисования граней. Например, сначала одну грань одного объекта, потом соответствующую грань другого объекта и так далее. Последовательность рисования в этом случае должна быть от самых дальних граней к самым близким. Однако это усложняет объектно-ориентированную реализацию программы, поскольку желательно было бы, чтобы объект был самодостаточным с точки зрения каждой операции, выполняемой над ним, — а это невозможно, так как операция сортировки граней должна обеспечивать доступ к отдельным граням, а не только к объекту в целом. Хотя объектно-ориентированная методология не накладывает столь жестких ограничений на реализацию объектов, однако, такое нарушение самодостаточности (инкапсуляции) выглядит не очень эстетично.

Поскольку среди элементов кроме многогранников есть сфера, то метод сортировки граней по глубине нельзя использовать, так как сфера — это не мно-

ногранник, и она рисуется по пикселам (хотя можно было бы определить ее как многогранник, закрашенный, например, по методу Гуро, однако это намного сложнее, и в данном примере программы не рассматривается). Необходимо использовать Z-буфер, а функция `Polygon` его не поддерживает. Более того, в составе функций API Windows нет ни одной функции рисования, рассчитанной на использование Z-буфера. Такие функции мы вынуждены сконструировать сами. Относительно объектной ориентированности — метод Z-буфера позволяет полностью инкапсулировать операцию рисования объекта в виде одной функции-члена (мы ее назовем `::Draw`). Один вызов функции `Draw` обеспечивает полный цикл отображения объекта соответствующего класса.

Текст программы (`studex34.cpp`):

```
//---------------(c)Copyright Порев В.Н.-------------------
#include "winmain1.cpp"
#include <math.h>
#define MaxObjectsNum 100
#define PERSPIKTIVE_VIEW_MODE 1

float *pZbuf = NULL;                //Z-буфер
long horSize,vertSize,SizeBuf;

double Matrix[12];          //матрица поворота
int cxView,cyView;

struct VERTEX               //тип описания 3D-точек
{
double x,y,z;
};

class Shape             //базовый абстрактный класс
{
protected:
  VERTEX centr;
  double size;
  BYTE red,gre,blu;
  void PrepareVertex(VERTEX *);
public:
  virtual void Move(double,double,double);
  virtual void ShapeSize(double);
  virtual void ShapeColor(BYTE,BYTE,BYTE);
  virtual void Draw(HDC) = 0;
};
```

```
class Sphere : public Shape
{
public:

  virtual void Draw(HDC);
};

class Pyramid : public Shape
{
public:
  virtual void Draw(HDC);
};

class Cube : public Shape
{
public:
  virtual void Draw(HDC);
};

class Column : public Cube
{
public:
  virtual void Draw(HDC);
};

class Area : public Cube
{
public:
  virtual void Draw(HDC);
};

void SetCameraViewMatrix(double alpha,double beta,
                         int cx,int cy);
void ViewTransform(VERTEX *v);
COLORREF ReflectionColor(COLORREF clr,
                         VERTEX v1,VERTEX v2,VERTEX v3);
void MyPolygon(HDC hdc,VERTEX *v,int nv,COLORREF clr);
void MyLineHor(HDC hdc,int x1,int x2,int y,
               double z1,double z2,COLORREF clr);
BOOL InitMyZbuffer(long cx, long cy);
void ClearMyZbuffer(void);
void SetPixMyZ(HDC hdc, int x, int y, float z,
               COLORREF clr);
```

```
void DrawStudyExample(HWND hWnd)
{
HDC hdc,hdcWin;
HBITMAP hBitmap;
HCURSOR hCurOld;
RECT rc;
int  objectIndex;
Shape *picture[MaxObjectsNum];      //массив объектов

hCurOld = SetCursor(LoadCursor(NULL,IDC_WAIT));
for (int i=0; i<MaxObjectsNum; i++)
    picture[i] = NULL;
picture[0] = new Area;     //создаем экземпляр объекта
picture[0] -> Move(0,0,-150);  //положение в пространстве
picture[0] -> ShapeSize(40);   //размер
picture[0] -> ShapeColor(0,255,0);  //цвет
objectIndex = 1;
for (int y=-200; y <= 200; y += 400)
  for (int x=-200; x <= 200; x += 400)
    {
    picture[objectIndex] = new Column;
    picture[objectIndex] -> Move(x,y,-80);
    picture[objectIndex] -> ShapeSize(20);
    picture[objectIndex] -> ShapeColor(255,255;0);
    objectIndex++;
    }
picture[5] = new Pyramid;
picture[5] -> Move(0,0,150);
picture[5] -> ShapeSize(250);
picture[5] -> ShapeColor(255,160,0);
picture[6] = new Sphere;
picture[6] -> Move(0,0,20);
picture[6] -> ShapeSize(125);
picture[6] -> ShapeColor(255,0,0);

hdcWin = GetDC(hWnd);
GetClientRect(hWnd,&rc);
hdc = CreateCompatibleDC(hdcWin);     //двойная буферизация
hBitmap = CreateCompatibleBitmap(hdcWin,rc.right,rc.bottom);
SelectObject(hdc,hBitmap);
InitMyZbuffer(rc.right,rc.bottom);
//-------Цикл одного полного оборота камеры----------------
for (int Alpha=10,Beta = 67; Alpha<=370; Alpha++,Beta++)
  {
  ClearMyZbuffer();
```

```
  PatBlt(hdc,0,0,rc.right, rc.bottom,WHITENESS);
  SetCameraViewMatrix(Alpha,Beta, rc.right, rc.bottom);
  for (int i=0; i<MaxObjectsNum; i++)  //цикл объектов
    if (picture[i])
       picture[i] -> Draw(hdc);    //рисуем объект
  BitBlt(hdcWin,0,0,rc.right,rc.bottom, hdc,0,0,SRCCOPY);
  }
DeleteDC(hdc);
DeleteObject(hBitmap);
for (int i=0; i<MaxObjectsNum; i++)  //уничтожение объектов
    delete []picture[i];
delete []pZbuf;              //закрытие Z-буфера
ReleaseDC(hWnd,hdcWin);
SetCursor(hCurOld);
}

void SetCameraViewMatrix(double alpha,double beta,
                         int cx,int cy)
{
alpha *= M_PI/180.0;
beta *= M_PI/180.0;
Matrix[0]  = cos(alpha);
Matrix[1]  = -sin(alpha);
Matrix[2]  = 0;
Matrix[3]  = 0;
Matrix[4]  = sin(alpha)*cos(beta);
Matrix[5]  = cos(alpha)*cos(beta);
Matrix[6]  = -sin(beta);
Matrix[7]  = 0;
Matrix[8]  = sin(alpha)*sin(beta);
Matrix[9]  = cos(alpha)*sin(beta);
Matrix[10] = cos(beta);
Matrix[11] = 0;
cxView = cx;     //будет использоваться для центрирования
cyView = cy;
}

void ViewTransform(VERTEX *v)
{
VERTEX tmp;

tmp.x = v->x * Matrix[0]
      + v->y * Matrix[1]
      + v->z * Matrix[2] + Matrix[3];
```

```
tmp.y = v->x * Matrix[4]
      + v->y * Matrix[5]
      + v->z * Matrix[6] + Matrix[7];
tmp.z = v->x * Matrix[8]
      + v->y * Matrix[9]
      + v->z * Matrix[10] + Matrix[11];

#if PERSPIKTIVE_VIEW_MODE
double zk = 2000, zpl = 600;

tmp.x *= (zk-zpl)/(zk-tmp.z); //для центральной проекции
tmp.y *= (zk-zpl)/(zk-tmp.z);
tmp.z -= zpl;
#endif

tmp.x += cxView/2;
tmp.y += cyView/2;
*v = tmp;
}

COLORREF ReflectionColor(COLORREF clr,
                         VERTEX v1,VERTEX v2,VERTEX v3)
{
double nx,ny,nz,k;

v2.x -= v1.x;
v2.y -= v1.y;
v2.z -= v1.z;
v3.x -= v1.x;
v3.y -= v1.y;
v3.z -= v1.z;
nx =  v2.y * v3.z - v3.y * v2.z;
ny = -v2.x * v3.z + v3.x * v2.z;
nz =  v2.x * v3.y - v3.x * v2.y;
k = 0.2 + 0.8*fabs(nz)/sqrt(nx*nx+ny*ny+nz*nz);
return RGB(k*(double)GetRValue(clr),
           k*(double)GetGValue(clr),
           k*(double)GetBValue(clr));
}

void MyPolygon(HDC hdc,VERTEX *v,int nv,COLORREF clr)
{
double x[4],z[4];
```

```
int i,y,nhor,min,max,st,en;
double x1,x2,y1,y2,z1,z2;

clr = ReflectionColor(clr, v[0], v[1], v[2]);
min = v[0].y;
max = v[0].y;
for (i=1; i<nv; i++)
  {
  if (min > v[i].y) min = v[i].y;
  if (max < v[i].y) max = v[i].y;
  }
for (y=min; y<=max; y++)
  {
  nhor = 0;
  for (i=0; i<nv; i++)
    {
    st = i;
    en = i+1;
    if (en >= nv) en=0;
    x1 = v[st].x;
    y1 = v[st].y;
    z1 = v[st].z;
    x2 = v[en].x;
    y2 = v[en].y;
    z2 = v[en].z;
    if ((y >= y1)&&(y < y2)||
        (y <= y1)&&(y > y2))
       if (y1 != y2)
          {
          x[nhor] = x1 + (x2-x1)*(double)(y-y1)
                                  /(double)(y2-y1);
          z[nhor] = z1 + (z2-z1)*(double)(y-y1)
                                  /(double)(y2-y1);
          nhor++;
          }
    }
  if (nhor == 2) MyLineHor(hdc,x[0],x[1],y,z[0],z[1],clr);
  }
}

void MyLineHor(HDC hdc,int x1,int x2,int y,
               double z1,double z2,COLORREF clr)
{
int x;
double k,z;
```

```
if (x1 == x2)
   {
   SetPixMyZ(hdc, x1, y, z1, 0);
   return;
   }
k = (z2-z1)/(double)(x2-x1);
if (x1 < x2)
   for (x=x1+1; x<x2; x++)
     {
     z = z1 + (double)(x - x1)*k;
     SetPixMyZ(hdc, x, y, z, clr);
     }
if (x1 > x2)
   for (x=x2+1; x<x1; x++)
     {
     z = z1 + (double)(x - x1)*k;
     SetPixMyZ(hdc, x, y, z, clr);
     }
SetPixMyZ(hdc, x1, y, z1, 0); //выделим ребра граней
SetPixMyZ(hdc, x2, y, z2, 0);
}
BOOL InitMyZbuffer(long cx, long cy)
{
pZbuf = new float [cx * cy];
if (pZbuf == NULL) return FALSE;
horSize = cx;
vertSize = cy;
return TRUE;
}
void ClearMyZbuffer(void)
{
long i,size;

if (pZbuf == NULL) return;
size = horSize * vertSize;
for (i=0; i<size; i++)
    pZbuf [i] = -10000;
}

void SetPixMyZ(HDC hdc, int x, int y, float z,
               COLORREF clr)
{
long indx;
```

```
if ((x<0)||(x>=horSize)||(y<0)||(y>=vertSize)) return;
indx = (long)y*horSize + (long)x;
if (z >= pZbuf [indx])
   {
   pZbuf [indx] = z;


   SetPixel(hdc,x,y,clr);
   }
}

//---------определение функций-членов классов--------------
void Shape::Move(double x,double y,double z)
{
centr.x = x;
centr.y = y;
centr.z = z;
}

void Shape::ShapeSize(double s)
{
size = s;
}

void Shape::ShapeColor(BYTE r,BYTE g,BYTE b)
{
red = r;
gre = g;
blu = b;
}

void Shape::PrepareVertex(VERTEX *v)
{
VERTEX tmp;

tmp.x = size * v->x + centr.x;
tmp.y = size * v->y + centr.y;
tmp.z = size * v->z + centr.z;
ViewTransform(&tmp);
*v = tmp;
}

void Sphere::Draw(HDC hdc)
{
COLORREF clr;
```

```
double R,R2,r2,z,k;
int x,y;
VERTEX c=centr;;

#if PERSPIKTIVE_VIEW_MODE
VERTEX v=centr;

v.z += size;
ViewTransform(&v);
ViewTransform(&c);
R2 = (c.x - v.x)*(c.x - v.x)
   + (c.y - v.y)*(c.y - v.y)
   + (c.z - v.z)*(c.z - v.z);
R = sqrt(R2);
#else
ViewTransform(&c);
R = size;
R2 = R*R;
#endif
for (x=0;x<=R;x++)
  for (y=0;y<=x;y++)
    {
    r2 = x*x + y*y;
    if (r2 > R2) break;
    z = sqrt(R2 - r2);
    k = 1 - r2/R2;
    clr = RGB((BYTE)(k*(double)red),
              (BYTE)(k*(double)gre),
              (BYTE)(k*(double)blu));
    SetPixMyZ(hdc,c.x + x, c.y + y, c.z + z, clr);
    SetPixMyZ(hdc,c.x + x, c.y - y, c.z + z, clr);
    SetPixMyZ(hdc,c.x - x, c.y + y, c.z + z, clr);
    SetPixMyZ(hdc,c.x - x, c.y - y, c.z + z, clr);
    SetPixMyZ(hdc,c.x + y, c.y + x, c.z + z, clr);
    SetPixMyZ(hdc,c.x + y, c.y - x, c.z + z, clr);
    SetPixMyZ(hdc,c.x - y, c.y + x, c.z + z, clr);
    SetPixMyZ(hdc,c.x - y, c.y - x, c.z + z, clr);
    }
}

void Pyramid::Draw(HDC hdc)
{
VERTEX v[5] = { {-1,1,0}, {1,1,0}, {1,-1,0},
                {-1,-1,0}, {0, 0, 0.5} };
```

```
char nv[5] = {4,3,3,3,3};  //количество вершин в гранях
char vindex[16] = {0,1,2,3, 0,1,4, 1,2,4, 2,3,4, 3,0,4};
VERTEX gran[4];
int pos = 0;

for (int i=0; i<5; i++)
    PrepareVertex(&v[i]);      //преобразование координат
for (int i=0; i<5; i++)
  {
  for (int j=0; j<nv[i]; j++)
    gran[j] = v[ vindex[pos+j] ];
  MyPolygon(hdc, gran, nv[i], RGB(red,gre,blu));
  pos += nv[i];
  }
}

void Cube::Draw(HDC hdc)
{
VERTEX v[8]={{-1, 1,-1}, {-1, 1,1}, {1, 1,1}, {1, 1,-1},
             {-1,-1,-1}, {-1,-1,1}, {1,-1,1}, {1,-1,-1}};
char vindex[24] = {0,1,2,3, 4,5,6,7, 0,1,5,4,
                   1,5,6,2, 2,6,7,3, 0,4,7,3};
VERTEX gran[4];
int pos = 0;

for (int i=0; i<8; i++)
  PrepareVertex(&v[i]);
for (int i=0; i<6; i++)
  {
  for (int j=0; j<4; j++)
    gran[j] = v[ vindex[pos+j] ];
  MyPolygon(hdc, gran, 4, RGB(red,gre,blu));
  pos += 4;
  }
}

void Column::Draw(HDC hdc)
{
double step=2.5*size;
VERTEX oldCentr = centr;

for (int i=0; i<5; i++)           //пять кубов
  {
  Move(oldCentr.x, oldCentr.y, oldCentr.z + i*step);
```

```
  Cube::Draw(hdc);
  }
centr = oldCentr;
}

void Area::Draw(HDC hdc)
{
double step=2.5*size;
VERTEX oldCentr = centr;

for (int j=-2; j<3; j++)          //25 кубов
  for (int i=-2; i<3; i++)
    {
    Move(oldCentr.x + i*step,
         oldCentr.y + j*step,
         oldCentr.z);
    Cube::Draw(hdc);
    }

centr = oldCentr;
}
```

Скомпилируйте и запустите программу `studex34`. Необходимо предупредить, что цикл показа может затянуться надолго. В программе выполняется полный оборот камеры на 360 градусов с шагом в один градус. Время создания и отображение всех 361 кадров в соответствующих ракурсах на компьютере с процессором AMD K6-2, 300 МГц, в 24-битном видеорежиме составляло 807 секунд. То есть, на один кадр расходуется в среднем 807/361 = 2.24 секунды. Размеры окна не изменялись после запуска программы, это отвечает размерам изображения 392 на 239 пикселов. Необходимо признать, что эта программа демонстрирует черепашью скорость рендеринга.

8.1. Анализ и оптимизация программы

Каждую программу можно усовершенствовать. Можно попробовать уменьшить текст программы, уменьшить размер выполняемого файла, улучшить структурированность, модульность и так далее. В данном случае мы попытаемся повысить скорость рендеринга — уменьшить время формирования кадров изображения.

Как оптимизировать программу по быстродействию? Для этого необходимо выполнить анализ работы программы. В результате анализа нужно обнару-

жить операции, которые обуславливают быстродействие программы. После того как будут найдены критические места программы, можно сделать выводы относительно конкретных путей оптимизации.

Для измерения времени выполнения операций в программе для Windows можно воспользоваться функцией API `GetLocalTime`:

```
SYSTEMTIME SysTimeStart,SysTimeEnd;
double sec;
char szNum[32];

GetLocalTime(&SysTimeStart);

. . . . . //а здесь расположены операции,
. . . . . //которые хронометрируются

GetLocalTime(&SysTimeEnd);
sec = (double)SysTimeEnd.wHour*3600
    + (double)SysTimeEnd.wMinute*60
    + (double)SysTimeEnd.wSecond
    + (double)SysTimeEnd.wMilliseconds/1000
    - (double)SysTimeStart.wHour*3600
    - (double)SysTimeStart.wMinute*60
    - (double)SysTimeStart.wSecond
    - (double)SysTimeStart.wMilliseconds/1000;
sprintf(szNum,"%. 2f",sec);
SetWindowText(hWnd,szNum);
```

Необходимо предупредить, что миллисекунды измеряются не очень точно, поэтому для повышения точности измерения для некоторой отдельной операции можно делать цикл из многих (сотен, тысяч, ...) одинаковых операций (если вспомогательные операции создания цикла сами по себе не длительные). Кроме того, различные сеансы измерений могут давать различные значения, поэтому необходимо как-то усреднять результаты. Понятно, что все измерения должны выполняться на одном и том же компьютере и обязательно в одинаковых условиях выполнения программы. Также необходимо учитывать, что в полночь измерение времени может дать ошибку, — если переход на 0 часов случится в ходе измерений. Впрочем, я и не рекомендую вам по ночам засиживаться за компьютером — ночью надо спать.

Теперь приступим к анализу программы `studex34`. Вся работа по созданию объектов, их отображение в различных ракурсах и уничтожение объектов делается в теле функции `DrawStudyExample`. Сделаем измерения времени ос-

ювных операций. На создание объектов, открытие контекста, подготовку битмапа двойного буфера и создание Z-буфера расходуется менее десяти миллисекунд (измерения с точностью до процентов секунд дают 0.00). Таким образом, в ходе дальнейшего анализа сосредоточимся на цикле создания 361 кадра.

```
for (int Alpha=10,Beta = 67; Alpha<=370; Alpha++,Beta++)
  {
  ClearMyZbuffer();
  PatBlt(hdc,0,0,rc.right, rc.bottom,WHITENESS);
  SetCameraViewMatrix(Alpha,Beta, rc.right, rc.bottom);
  for (int i=0; i<MaxObjectsNum; i++)
      if (picture[i])
         picture[i] -> Draw(hdc);
  BitBlt(hdcWin,0,0,rc.right,rc.bottom, hdc,0,0,SRCCOPY);
  }
```

Как измерить время, расходуемое во всех 361 кадрах на выполнение функции `ClearMyZbuffer()`? Это сделаем способом, который можно назвать "способом контрольно-измерительного стенда". Такой "стенд" можно сделать на основе текста нашей программы, например, следующим образом:

```
GetLocalTime(&SysTimeStart);
for (int j=10; j<=370; j++)
  {
  ClearMyZbuffer();
  }
GetLocalTime(&SysTimeEnd);
. . .
//далее вычисляем разность в секундах и выводим результат
```

Разумеется, подобный способ измерений можно считать корректным лишь тогда, когда время выполнения функции `ClearMyZbuffer()` значительно больше, чем время выполнения операций организации цикла по `j`.

Время выполнения 361 операции `ClearMyZbuffer` составляет в среднем 2.1 секунды. Аналогично можно сделать измерения для `PatBlt` — 0.06 сек., `SetCameraViewMatrix` — 0.00 сек., `BitBlt` — 0.5 сек. Однако делать измерения времени для цикла отображения объектов таким "стендовым" способом нельзя. Для корректного создания изображения обязательно выполнение всех подготовительных операций в полном объеме. Для измерений времени здесь можно предложить другой способ. Суть его такова. Вначале измеряем время выполнения полного цикла создания изображений:

```
for (int Alpha=10,Beta = 67; Alpha<=370; Alpha++,Beta++)
  {
  ClearMyZbuffer();
  PatBlt(hdc,0,0,rc.right, rc.bottom,WHITENESS);
  SetCameraViewMatrix(Alpha,Beta, rc.right, rc.bottom);
  for (int i=0; i<MaxObjectsNum; i++)
      if (picture[i])
         picture[i] -> Draw(hdc);
  BitBlt(hdcWin,0,0,rc.right,rc.bottom, hdc,0,0,SRCCOPY);
  }
```

А потом исключаем анализируемую операцию:

```
for (int Alpha=10,Beta = 67; Alpha<=370; Alpha++,Beta++)
  {
  ClearMyZbuffer();
  PatBlt(hdc,0,0,rc. right, rc. bottom,WHITENESS);
  SetCameraViewMatrix(Alpha,Beta, rc. right, rc. bottom);
// for (int i=0; i<MaxObjectsNum; i++)
//      if (picture[i])
//         picture[i] -> Draw(hdc);
  BitBlt(hdcWin,0,0,rc.right,rc.bottom, hdc,0,0,SRCCOPY);
  }
```

и измеряем время выполнения без нее. Полный цикл — 807 сек., без исключенной операции — 2.6 сек. Назовем такой способ "временным исключением". Необходимо заметить, что цифру 2.6 можно было бы получить и иначе, если от времени полного цикла вычесть уже измеренное время других операций. Однако способ "временного исключения" предназначен в первую очередь для тех случаев, когда измерение всех составных операций затруднено или не нужно. Продолжим измерения дальше.

Как мы видим, в цикле создания кадров на подготовительные операции расходуется мало времени в сравнении с отображением объектов. В ходе отображения объектов выполняется много операций. Какая из них самая длительная? Осуществим поиск способом "временного исключения". Однако для применения этого способа есть много ограничений. Сформулируем основные условия корректного использования данного способа.

1. Временно исключить можно не любую операцию, а только ту, отсутствие которой не нарушает логику работы программы. Другие операции должны выполняться в полном объеме и в той же последовательности. Это главное условие. Остальные условия можно сформулировать как следствие.

2. Необходимо быть внимательными, если вы работаете с оптимизирующим компилятором. Он может сделать такие изменения в программе во время компиляции, о которых мы и не подозреваем. Мы можем исключить одну операцию — а компилятор исключит еще несколько и никак нам об этом не сообщит. Это приведет к иллюзии значительной роли временно исключенной операции.

3. Нельзя изменять стратегию использования виртуальной памяти (если это специально не анализируется). Например, в функции `DrawStudyExample` нами не используется ни одна из файловых операций. Однако в ходе выполнения этой функции, программа может часто обращаться к диску. Это может быть в случаях, когда открываются значительные по объему массивы, и операционная система делает перераспределение виртуальной памяти между оперативной памятью (RAM) и диском. А в данное время мы временно выключили эту операцию, и обращения к диску прекратились — это может быть свидетельством того, что отныне все массивы целиком размещаются в RAM. Последнее может привести к ускорению выполнения программы, поскольку обращение к RAM осуществляется намного быстрее, чем к диску. Кроме того, когда размера RAM недостаточно для полной программы, то даже уменьшение объема кода при исключении может привести к тому, что программа будет работать быстрее — так как отныне все размещается в RAM. Однако в этом случае уменьшения времени не пропорционально времени выполнения исключенной функции, и это не позволит рассчитать ее вклад в общее время выполнения программы. Таким образом, необходимо пользоваться правилом: объем оперативной памяти должен быть достаточным как для полной программы, так и программы с исключением.

Способ временного исключения отдельных операций можно трактовать так: **если после исключения некоторой операции время выполнения программы уменьшается не существенно, то эту операцию не нужно оптимизировать в первую очередь**. Однако если в результате анализа мы и обнаружим некоторую длительную операцию, то это еще не означает, что ее можно ускорить.

Продолжим дальше анализ программы. Временно выключить функцию `MyPolygon` нельзя, поскольку тогда будет рисоваться только шар, а это означает другой порядок заполнения пикселами Z-буфера и растра битмапа двойного буфера. По аналогичной причине нельзя временно выключить функцию `Sphere::Draw(HDC hdc)`. А что же тогда можно?

Рассмотрим функцию `SetPixMyZ`. Если ее исключить, то прекратится запись пикселов в оба растра — Z-буфера и растра битмапа. Однако те функции, ко-

торые остались, выполняются так же, как и до исключения. Временно исключим SetPixMyZ. Результат — 57.5. То есть, из 807 секунд почти все время расходуется на запись пикселов.

Составная часть функции SetPixMyZ — вызовы функции SetPixel. Временно исключим ее. Результат — 70 секунд. Результаты временных исключений отобразим следующей диаграммой (рис. 8.3).

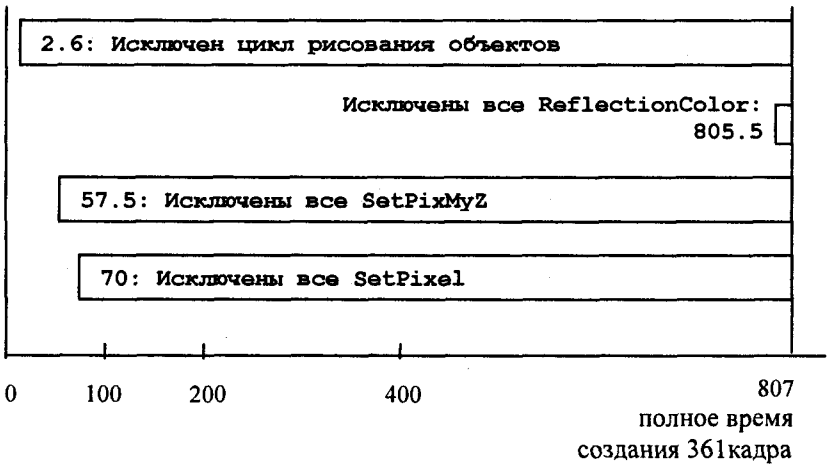


Рис. 8.3. Диаграмма исключения операций

По диаграмме исключения можно рассчитать, сколько времени расходуется в программе на выполнение отдельных операций:

$$T\ операции = T\ полное - T\ при\ исключении\ этой\ операции.$$

Время выполнения функции SetPixel:

$$T(\texttt{SetPixel}) = T полное - T искл.\ \texttt{SetPixel} = 807 - 70 = 737 \text{ секунд.}$$

Необходимо учитывать, что функция SetPixel — вложенная по отношению к SetPixMyZ.

```
void SetPixMyZ(HDC hdc,int x,int y,float z,COLORREF clr)
{
long indx;

if ((x<0)||(x>=horSize)||(y<0)||(y>=vertSize)) return;
indx = (long)y*horSize + (long)x;
```

```
if (z >= pZbuf [indx])
   {
   pZbuf [indx] = z;
   SetPixel(hdc,x,y,clr);
   }
}
```

поэтому целесообразно будет для анализа `SetPixMyZ` рассматривать долю времени, не относящегося к вызову `SetPixel`:

$$T(\texttt{SetPixMyZ}) = T_{полное} - T_{искл.}\texttt{SetPixMyZ} - T(\texttt{SetPixel}) = \\ = 807 - 57.5 - 737 = 12.5 \text{ секунд.}$$

Таким образом, мы уже можем рассмотреть результаты измерений для некоторых отдельных операций:

`SetPixel`	–737;
`SetPixMyZ (без SetPixel)`	–12.5;
`ClearMyZbuffer`	–2.1;
`ReflectionColor`	–1.5;
`PatBlt`	–0.06;
`SetCameraViewMatrix`	–0.00;
`BitBlt`	–0.5 секунды.

Этот перечень неполон, можно анализировать еще некоторые функции, однако уже ясно, что основная причина низкой скорости — это использование функции `SetPixel`. Ее мы никак не можем изменить, ибо это функция API Windows. Но можно попробовать обойтись без нее.

Один из способов работы с растром — непосредственный доступ к памяти, хранящей растровый массив. Такой способ достаточно известен. Он использовался при разработке почти всех быстрых графических программ для когда-то популярной операционной среды MS-DOS. Среди функций MS-DOS предусмотрена функция рисования пиксела на экране, но она работала так же медленно, как и функция API Windows `SetPixel`. Поэтому для создания изображений на экране часто использовались операции непосредственной записи в видеопамять. В операционной системе Windows обращение прикладных программ к видеопамяти запрещено, а вся растровая графика основывается на понятии контекста графического устройства. Через контекст мы рисуем на экране, через контекст — в растрах битмапов. Кажется, это и все. Однако разработчики Windows предусмотрели еще одну возможность работы с растрами — операции с растрами в формате **DIB** (Device Independent Bitmap).

Можно создавать растр, не привязанный к какому-то графическому устройству по формату пикселов, а самое главное — предусмотрен доступ к растру по указателю на массив в памяти. С этим массивом можно производить любые операции, поскольку становится известным его адрес в виртуальной памяти. В том числе и выполнять операции над отдельными байтами и битами, которые представляют пикселы растра. Это открывает широкие возможности для создания собственных графических библиотек.

Текст программы studex35.cpp:

```
//-----------------(c)Copyright Порев В.Н.----------------
#include "winmain1.cpp"
#include <math.h>
#define MaxObjectsNum 100
#define PERSPIKTIVE_VIEW_MODE 1

float *pZbuf = NULL;                    //Z-буфер
long horSize,vertSize,SizeBuf;

BITMAPINFO *pBitInfo=NULL;
BYTE *pRastBuf=NULL;                    //растровый буфер (DIB)
long bytesPerString,HorBuf,VertBuf;

double Matrix[12];
int cxView,cyView;

struct VERTEX
{
double x,y,z;
};

//--объявление всех классов так же, как в studex35.cpp---
class Shape
. . .
class Sphere : public Shape
. . .
class Piramid : public Shape
. . .
class Cube : public Shape
. . .
class Column : public Cube
. . .
class Area : public Cube
. . .
```

```
void SetCameraViewMatrix(double alpha,double beta,
                         int cx,int cy);
void ViewTransform(VERTEX *v);
COLORREF ReflectionColor(COLORREF clr,
                         VERTEX v1, VERTEX v2, VERTEX v3);
void MyPolygon(VERTEX *v,int nv,COLORREF clr);
void MyLineHor(int x1,int x2,int y,
               double z1,double z2,COLORREF clr);
BOOL InitMyZbuffer(long cx, long cy);
void ClearMyZbuffer(void);
void SetPixMyZ(int x, int y, float z, COLORREF clr);
BOOL OpenRastrMem(long cx,long cy);
void CloseRastrMem(void);
void ClearRastrMem(void);
void SetPixRastrMem(long x, long y, COLORREF clr);

void DrawStudyExample(HWND hWnd)
{
HDC hdcWin;
HCURSOR hCurOld;
RECT rc;
int  objectIndex;
Shape *picture[MaxObjectsNum];

hCurOld = SetCursor(LoadCursor(NULL,IDC_WAIT));
for (int i=0; i<MaxObjectsNum; i++)
    picture[i] = NULL;

picture[0] = new Area;
picture[0] -> Move(0,0,-150);
picture[0] -> ShapeSize(40);
picture[0] -> ShapeColor(0,255,0);
objectIndex = 1;
for (int y=-200; y <= 200; y += 400)
    for (int x=-200; x <= 200; x += 400)
        {
        picture[objectIndex] = new Column;
        picture[objectIndex] -> Move(x,y,-80);
        picture[objectIndex] -> ShapeSize(20);
        picture[objectIndex] -> ShapeColor(255,255,0);
        objectIndex++;
        }
picture[5] = new Piramid;
picture[5] -> Move(0,0,150);
```

```
picture[5] -> ShapeSize(250);
picture[5] -> ShapeColor(255,160,0);
picture[6] = new Sphere;
picture[6] -> Move(0,0,20);
picture[6] -> ShapeSize(125);
picture[6] -> ShapeColor(255,0,0);

hdcWin = GetDC(hWnd);
GetClientRect(hWnd,&rc);
InitMyZbuffer(rc.right,rc.bottom);
OpenRastrMem(rc.right,rc.bottom);     //открываем DIB
for (int Alpha=10,Beta = 75; Alpha<=370; Alpha++,Beta++)
    {
    ClearMyZbuffer();
    ClearRastrMem();
    SetCameraViewMatrix(Alpha,Beta, rc.right, rc.bottom);
    for (int i=0; i<MaxObjectsNum; i++)
        if (picture[i])
           picture[i] -> Draw(hdcWin);
    StretchDIBits(hdcWin,0,0, rc.right, rc.bottom,
                         0,0, rc.right, rc.bottom,
                         pRastBuf,pBitInfo,
                         DIB_RGB_COLORS,SRCCOPY);
    }
for (int i=0; i<MaxObjectsNum; i++)
    delete []picture[i];
CloseRastrMem();              //закрываем DIB
delete []pZbuf;
ReleaseDC(hWnd,hdcWin);
SetCursor(hCurOld);
}

//-------------определение этих функций-------------
//-------полностью совпадает с studex35.cpp---------
void SetCameraViewMatrix(double alpha,double beta,
                         int cx,int cy)
. . .
void ViewTransform(VERTEX *v)
. . .
COLORREF ReflectionColor(COLORREF clr,
                         VERTEX v1, VERTEX v2, VERTEX v3)
. . .
void MyPolygon(VERTEX *v,int nv,COLORREF clr)
. . .
```

```
void MyLineHor(int x1,int x2,int y,
               double z1,double z2,COLORREF clr)
. . .
BOOL InitMyZbuffer(long cx, long cy)
. . .
void ClearMyZbuffer(void)
. . .
//---------открываем DIB в формате 24 бит/пиксел-------
BOOL OpenRastrMem(long hor,long vert)
{
long bmWidth;

pBitInfo  = new BITMAPINFO [sizeof(BITMAPINFO)];
if (pBitInfo == NULL) return FALSE;
bmWidth = hor & 0xfffc;              //должно быть кратно 4
if (bmWidth < hor) bmWidth += 4;
bmWidth += 8;
bytesPerString = 3*bmWidth;
HorBuf = hor;
VertBuf = vert;
SizeBuf = bytesPerString*VertBuf;
pBitInfo->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
pBitInfo->bmiHeader.biWidth = bmWidth;
pBitInfo->bmiHeader.biHeight = vert;
pBitInfo->bmiHeader.biPlanes = 1;
pBitInfo->bmiHeader.biBitCount = 24;
pBitInfo->bmiHeader.biClrUsed = 0;
pBitInfo->bmiHeader.biClrImportant = 0;
pBitInfo->bmiHeader.biCompression = BI_RGB;
pRastBuf = new BYTE[SizeBuf];
if (pRastBuf == NULL)
   {
   delete []pBitInfo;
   return FALSE;
   }
return TRUE;
}

void CloseRastrMem(void)
{
delete []pRastBuf;
delete []pBitInfo;
}
```

```
void ClearRastrMem(void)
{
memset(pRastBuf,255,SizeBuf);  //заполняем белым цветом
}

void SetPixMyZ(int x, int y, float z, COLORREF clr)
{
long indx;

if ((x<0) || (x>=horSize) || (y<0) || (y>=vertSize)) return;
indx = (long)y*horSize + (long)x;
if (z >= pZbuf [indx])
   {
   pZbuf [indx] = z;
   SetPixRastrMem(x, y, clr);
   }
}

//--------запись одного пиксела в растр DIB-------------
void SetPixRastrMem(long x, long y, COLORREF clr)
{
long lKoord;

lKoord  = bytesPerString * (VertBuf-y-1) + 3*x;
pRastBuf[lKoord]   = GetBValue(clr);
pRastBuf[lKoord+1] = GetGValue(clr);
pRastBuf[lKoord+2] = GetRValue(clr);
}

//-------определение всех функций-членов классов-----
//-------далее полностью совпадает с studex35.cpp----
void Shape::Move(double x,double y,double z)
. . . .
. . . .
```

В программе использована одна из функций API Windows для работы с растрами DIB. Это функция `StretchDIBits`, которая осуществляет вывод растра из памяти по адресу `pRastBuf` в определенный контекст. Растровый буфер DIB здесь в формате 24-битного цвета — это позволяет достаточно просто моделировать различные интенсивности отражения света для цветных объектов. Для корректности сравнения работы обеих программ (`studex34,35`) все испытания следует производить в видеорежиме True Color 24 бит на пиксел.

В 24-битном режиме цвет каждого пиксела определяется тройкой байтов RGB. Как раз эти байты и записываются в память функцией `SetPixRastrMem`.

Скомпилируйте и проверьте работу программы `studex35`. Полный оборот камеры в ней делается за 88 секунд в отличие от `studex34`, где полный оборот составлял 807 секунды. Таким образом, создание одного кадра выполняется в среднем за 88/361 = 0.24 секунды. Благодаря замене функции `SetPixel` нашей собственной функцией `SetPixRastrMem` достигнуто уменьшение времени рендеринга более, чем в 9 раз. И это несмотря на то, что функция `StretchDIBits` работает медленнее, чем `BitBlt`.

Необходимо отметить, что функция `SetPixRastrMem` предназначена только для 24-битных растров, а `SetPixel` корректно работает во всех цветовых форматах, поэтому она и работает медленнее.

8.2. И снова анализ и оптимизация программы

Проанализируем теперь программу `studex35`. Для нее диаграмма исключения операций имеет следующий вид (рис. 8.4).

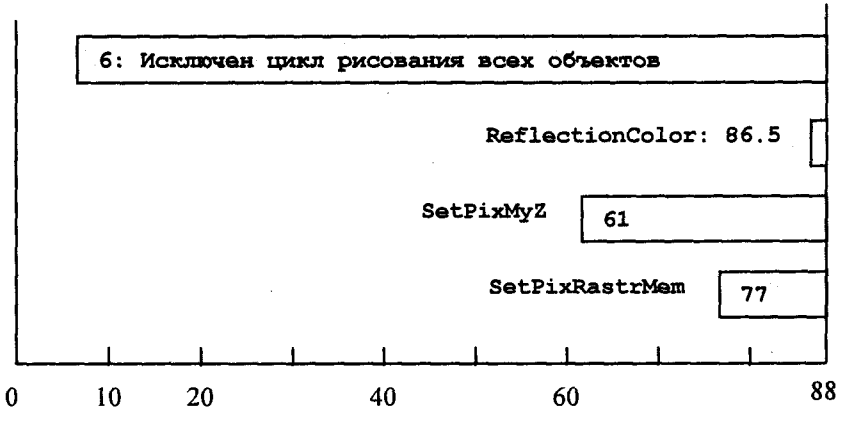


Рис. 8.4. Диаграмма исключения операций `studex35`

Как мы видим, операции записи пикселов в растр DIB составляют уже небольшую часть общего цикла рисования. Также уменьшилась доля времени для записи всех пикселов (`SetPixMyZ`). Необходимо проанализировать другие операции.

Ранее, в ходе анализа программы `studex34`, мы отмечали, что нельзя временно исключать операцию `MyPolygon`, так как изменится логика заполнения

Z-буфера — кроме полигонов в нашей программе рисуется и шар. И все же давайте сделаем отдельный анализ для цикла вывода многогранников, исключив на весь период измерений функцию `Sphere::Draw`. Понятно, что без шара процесс создания изображения будет проистекать по-другому. Однако мы вынуждены пойти на это — когда сложно анализировать весь процесс в целом, необходимо попробовать разделить его на более простые составляющие.

Снова воспользуемся методом временного исключения. Замеры времени отражены на следующей диаграмме исключения для многогранников (рис. 8.5).

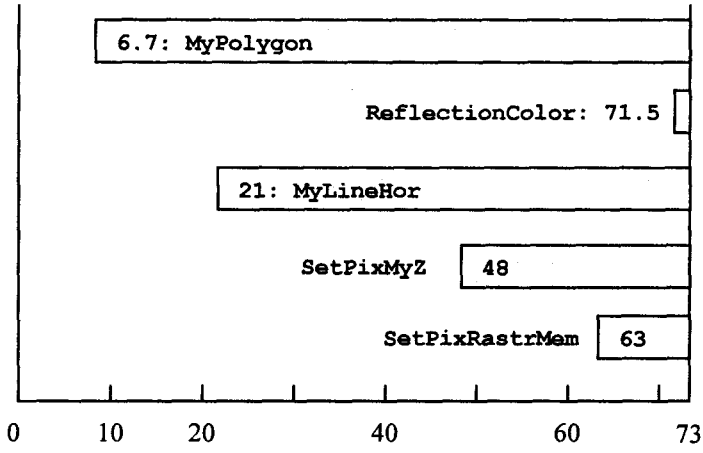


Рис. 8.5. Операции вывода многогранников

Определим общее время, затрачиваемое для работы функций:

```
MyPolygon          73 - 6.7  =  66.3
ReflectionColor    73 - 71.5 =   1.5
MyLineHor          73 - 21   =  52

SetPixMyZ          73 - 48   =  25
SetPixRastrMem     73 - 63   =  10 секунд.
```

Глядя на эти цифры, можно предположить, что необходимо оптимизировать функцию `MyPolygon`, которая вносит наибольший вклад в общее время. Однако, для того чтобы выяснить, как именно ее оптимизировать, следует проанализировать структуру программы, а точнее — иерархию вложенных функций. Вложенной функцией мы будем для краткости называть функцию, которая вызывается в теле другой функции. Таким образом, общее время работы

некоторой функции складывается из вызовов вложенных функций (если они есть) плюс время работы операторов собственно этой функции

$$T \text{ функции общее} = T \text{ собственное} + T \text{ вложенных функций}.$$

Для анализа каждой функции будем в первую очередь оценивать время собственной работы, а не время работы вложенных функций. Это время, очевидно, равно

$$T \text{ собственное} = T \text{ функции общее} - T \text{ вложенных функций}.$$

На рис. 8.6 изображена иерархия вложенности функций `studex35`.

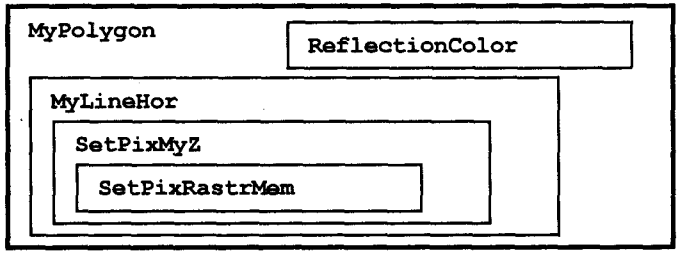


Рис. 8.6. Вложенность функций

Из этой схемы видно, что собственное время работы функции `MyPolygon` составляет разность между общим временем работы этой функции и временем работы функций `ReflectionColor` и `MyLineHor`:

$$T \text{ собственное}(\texttt{MyPolygon}) = 66.3 - 1.5 - 52 = \mathbf{12.8}.$$

Для остальных функций собственное время составит

MyLineHor = 52 - 25 = **27**

SetPixMyZ = 25 - 10 = **15** секунд.

Обратим внимание на эти цифры. Подозрительно много времени занимают операции тела функции `MyLineHor`. Возможно, ее следует оптимизировать в первую очередь. Следующим в этом списке фигурирует тело функции `SetPixMyZ` — 15 сек. Собственное время работы операторов тела функции `MyPolygon` относительно невелико.

План оптимизации программы может быть таким:

1. Совершенствование функции `MyLineHor`.
2. Уменьшение количества уровней вложения функций — можно сэкономить время, которое расходуется на вызовы вложенных функций и пере-

дачу им значений аргументов. Для этого код функции `SetPixRastrMem` расположим непосредственно в теле функции `SetPixMyZ`. Отдельная функция `SetPixRastrMem` нам уже не нужна.

Приведем текст третьего варианта программы (`studex36`) только для оптимизированных функций. Другие функции остались без изменений.

Часть файла `studex36.cpp`:

```
//---------------(c)Copyright Порев В.Н.-----------------
//---------горизонталь заполнения полигона---------------
void MyLineHor(int x1, int x2, int y,
               float z1, float z2, COLORREF clr)
{
int x;
float k,z,zz;
long indx,lKoord;
BYTE r,g,b;

if ((y < 0)||(y >= VertBuf)) return;
if (x1 == x2)
   {
   SetPixMyZ(x1, y, z1, 0);
   return;
   }
if (x1 > x2)
   {
   x = x1;
   x1 = x2;
   x2 = x;
   z = z1;
   z1 = z2;
   z2 = z;
   }
if (x1 >= HorBuf) return;
if (x2 < 0) return;
r = GetRValue(clr);
g = GetGValue(clr);
b = GetBValue(clr);
k = (z2-z1)/(float)(x2-x1);
zz = z1 - (float)x1*k;
indx = (long)y*horSize + (long)x1;
lKoord  = bytesPerString * (VertBuf-y-1) + 3*x1;
```

```
for (x=x1+1; x<x2; x++)
    {
    indx ++;
    lKoord += 3;
    if (x >= HorBuf) break;
    if (x>=0)
       {
       z = zz + (float)x*k;
       if (z >= pZbuf [indx])
          {
          pZbuf [indx] = z;
          pRastBuf[lKoord]   = b;
          pRastBuf[lKoord+1] = g;
          pRastBuf[lKoord+2] = r;
          }
       }
    }
SetPixMyZ(x1, y, z1, 0);
SetPixMyZ(x2, y, z2, 0);
}

void SetPixMyZ(int x, int y, float z, COLORREF clr)
{
long indx,lKoord;

if ((x<0) || (x>=horSize) || (y<0) || (y>=vertSize))
   return;
indx = (long)y*horSize + (long)x;
if (z >= pZbuf [indx])

   {
   pZbuf [indx] = z;
   lKoord  = bytesPerString * (VertBuf-y-1) + 3*x;
   pRastBuf[lKoord]   = GetBValue(clr);
   pRastBuf[lKoord+1] = GetGValue(clr);
   pRastBuf[lKoord+2] = GetRValue(clr);
   }
}
```

В результате второго шага оптимизации программа осуществляет полный цикл вывода многогранников за 49 секунд (вместо 73 для предыдущего варианта). Замена типов `double` на `float` дает немного — где-то одну-две секунды. Повышения скорости в основном достигнуто за счет усовершенствования

функции `MyLineHor` и включения кода записи пикселов DIB в тело функции `SetPixMyZ`. Функция `SetPixRastrMem` ликвидирована (ее делать `inline` не имеет смысла).

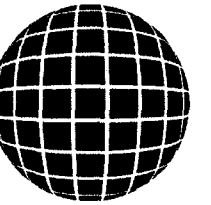
Восстановим работу программы в полном объеме — разблокируем `Sphere::Draw` и сделаем измерения времени. Полный цикл рисования составляет 65 секунд (предыдущий вариант был 88 секунд).

Таким образом, мы прошли два шага от первого варианта (`studex34`) и уменьшили время графического вывода от 807 до 65 секунд — то есть более чем в 12 раз. **Оптимизировать можно бесконечно любую программу**, в том числе и последний вариант. Например, использовать ассемблер для программирования некоторых критических функций. Однако это тема уже для другого разговора...

Главное, что здесь желательно увидеть, — мы находились в рамках только одной технологии создания графических программ (программирование на основе функций API Windows), хотя результаты программирования могут существенно отличаться.

Подобная методика анализа может использоваться не только для графических программ.

ГЛАВА 9



Пример анимации

Анимация — это создание иллюзии движения, изменения чего-то во времени. Для этого генерируется последовательность кадров путем моделирования развития во времени определенных процессов. В предыдущих главах в некоторых примерах программ мы уже использовали такой способ показа, например, обзор движущейся камерой неподвижных объектов (см. главу 8).

Морфингом (***morphing***) называются методы моделирования изменений формы объекта. Обычно стадии преобразования формы объекта определяются с помощью множества опорных точек. Потом выполняется некоторая интерполяция и показ всех этапов трансформации [47].

9.1. Поверхность Безье

Рассмотрим пример "живой" поверхности Безье, задаваемой точками-ориентирами. Если плавно перемещать в пространстве эти точки-ориентиры, то поверхность Безье будет соответственно видоизменяться. Выберем кубическую поверхность Безье.

Для описания изменения расположения точек-ориентиров используем пространственные кубические кривые Безье, заданные в параметрической форме. Параметром будет время (или номер кадра). Для описания формы кубических кривых требуется четыре опорные точки.

Таким образом, тут используются кривые Безье для описания стадий трансформаций и поверхности Безье в качестве объектов трансформации. Можно это назвать вариацией на тему Безье. Кроме того, здесь мы попрактикуемся в закрашивании поверхностей методом Гуро.

Файл studex40.cpp:

```
//-------------Цветок (c)Copyright Порев В.Н.-------------
#include "winmain1.cpp"
#include <math.h>

struct VERTEX
{
double x,y,z;
};
double Aview,Bview,Cview,Dview,Eview,
       Fview,Gview,Hview,Xview,Yview;
float *pZbuf = NULL;                    //Z-буфер
long horSize,vertSize,SizeBuf;

BITMAPINFO *pBitInfo=NULL;              //буфер DIB
BYTE *pRastBuf=NULL;
long bytesPerString,HorBuf,VertBuf;

double Px[4][4];       //опорные точки поверхности Безье
double Py[4][4];
double Pz[4][4];

//PPx, Ppy, PPz — массивы опорных точек трансформации.
//Четверки точек описывают кривые Безье, которые, в свою
//очередь, определяют точки-ориентиры Pij поверхности Безье
double PPx[64] =
 {  1,   1,   1,   1,                   //P00
   80,  80,  80,  80,                   //P01
   80,  80,  80, 256,                   //P02
    1, 100, 200, 367,                   //P03
    2,   2,   2,   2,                   //P10
  120, 120, 120, 120,                   //P11
  120, 120, 120, 201,                   //P12
    2, 100, 200, 300,                   //P13
    2,   2,   2,   2,                   //P20
  120, 120, 120, 120,                   //P21
  120, 120, 120, 189,                   //P22
    2,  80, 160, 262,                   //P23
    1,   1,   1,   1,                   //P30
   80,  80,  80,  80,                   //P31
   80,  80,  80, 174,                   //P32
    1,  60, 160, 240};                  //P33
```

```
double PPy[64] =
 { -1, -1, -1, -1,
  -80, -80, -80, -80,
  -80, -80, -80, -76,
   -1, -30, -60,-131,
   -1, -1, -1, -1,
  -40, -40, -40, -40,
  -40, -40, -40, -36,
   -1, 0, 2, 3,
    1, 1, 1, 1,
   40, 40, 40, 40,
   40, 40, 40, 37,
    1, 10, 20, 29,
    1, 1, 1, 1,
   80, 80, 80, 80,
   80, 80, 80, 76,
    1, 10, 20, 87};

double PPz[64] =
 { 0, 0, 0, 0,
    0, 0, 0, 0,
  150, 150, 150, 117,
  150, 150, 150, 36,
    0, 0, 0, 0,
    0, 0, 0, 0,
  150, 150, 150, 150,
  150, 150, 150, 68,
    0, 0, 0, 0,
    0, 0, 0, 0,
  150, 150, 150, 150,
  150, 150, 150, 47,
    0, 0, 0, 0,
    0, 0, 0, 0,
  150, 150, 150, 89,
  150, 150, 150, 0};

void DefineSurfacePoints(double t,double ang);
void DrawBezierSurface(COLORREF clr);
void BezierPoint(VERTEX *v,double s, double t);
void InitViewTranform(double alpha, double beta,
                      int cx, int cy);
void ScreenKoord(VERTEX *v);
double NormalVectorCos(VERTEX v1,VERTEX v2,VERTEX v3);
```

```
BOOL InitMyZbuffer(long cx, long cy);
void ClearMyZbuffer(void);
void SetPixMyZ(int x, int y, float z, COLORREF clr);
BOOL OpenRastrMem(long cx,long cy);
void CloseRastrMem(void);
void ClearRastrMem(void);
void MyPolygonGouraud(VERTEX *v,double *kr,
                      int nv,COLORREF clr);
void MyLineHorGouraud(int x1,int x2,int y,
                      double z1,double z2,
                      double k1,double k2,COLORREF clr);
void SetPixRastrMem(long x, long y, COLORREF clr);

void DrawStudyExample(HWND hWnd)
{
RECT rc;
HDC hdc;
double t,t1,t2;
COLORREF clr;

hdc = GetDC(hWnd);
GetClientRect(hWnd,&rc);
InitViewTranform(27, 50, rc.right, 11*rc.bottom/10);
PatBlt(hdc,0, 0, rc.right, rc.bottom, WHITENESS);
InitMyZbuffer(rc.right,rc.bottom);
OpenRastrMem(rc.right,rc.bottom);
for (int i=0; i<=100; i++)
    {
    t = (double)i/100.0;
    ClearMyZbuffer();
    ClearRastrMem();
    clr = RGB((1-0.3*t)*255, 0, (1-0.3*t)*128);
    for (int a=0;a<360;a+=90)
        {
        DefineSurfacePoints(t,a);
        DrawBezierSurface(clr);
        }    t1 = t-0.2;
    if (t1 < 0) t1=0;
    clr = RGB((1-0.3*t1)*255, 0, (1-0.3*t1)*128);
    for (int a=45;a<360;a+=90)
        {
        DefineSurfacePoints(t1,a);
        DrawBezierSurface(clr);
        }
    t2 = t-0.6;
```

```
    if (t2 < 0) t2=0;
    clr = RGB(255,0,128);
    if (t1 > 0)
       for (int a=0;a<360;a+=90)
           {
           DefineSurfacePoints(t2,a);
           DrawBezierSurface(clr);
           }
    StretchDIBits(hdc,0,0, rc.right, rc.bottom,
                      0,0, rc.right, rc.bottom,
                      pRastBuf,pBitInfo,
                      DIB_RGB_COLORS,SRCCOPY);
    }
CloseRastrMem();
delete []pZbuf;
ReleaseDC(hWnd,hdc);
}

//--расчет текущих точек-ориентиров поверхности Безье--
void DefineSurfacePoints(double t,double ang)
{
int i,j,ii,jj,index;
double tmpX[4],tmpY[4],tmpZ[4];
double A,B;

ang *= M_PI/180;
A = cos(ang);
B = sin(ang);
for (i=0;i<4;i++)
  for (j=0;j<4;j++)
      {
      for (ii=0;ii<4;ii++)
          {
          index = i*16 + j*4 + ii;
          tmpX[ii] =  A*PPx[index] + B*PPy[index];
          tmpY[ii] = -B*PPx[index] + A*PPy[index];
          tmpZ[ii] = PPz[index];
          }
      for (jj=3;jj>0;jj--)
        for (ii=0;ii<jj;ii++)
            {
            tmpX[ii] = tmpX[ii] + t*(tmpX[ii+1]-tmpX[ii]);
            tmpY[ii] = tmpY[ii] + t*(tmpY[ii+1]-tmpY[ii]);
            tmpZ[ii] = tmpZ[ii] + t*(tmpZ[ii+1]-tmpZ[ii]);
            }
```

```
      Px[i][j] = tmpX[0];
      Py[i][j] = tmpY[0];
      Pz[i][j] = tmpZ[0];
      }
}

void DrawBezierSurface(COLORREF clr)
{
int i,j,num;
VERTEX gran[4];
VERTEX vb[11][11];     //координаты узлов поверхности
VERTEX v;
double kgran[4];
double kr[11][11];     //коэффициенты отражения света
double kn;

for (i=0;i<11;i++)
  for (j=0;j<11;j++)
      {
      BezierPoint(&v, (double)i/10, (double)j/10);
      ScreenKoord(&v);
      vb[i][j] = v;
      }
//затем для всех вершин вычисляем косинус угла нормали
//--------для моделирования диффузного отражения-------
for (i=0;i<11;i++)
  for (j=0;j<11;j++)
      {
      num = 0;
      kn = 0;
      if ((i<10)&&(j<10))
         {
         kn += NormalVectorCos(vb[i][j],
                               vb[i+1][j],
                               vb[i][j+1]);
         num++;
         }
      if ((i<10)&&(j>0))
         {
         kn += NormalVectorCos(vb[i][j],
                               vb[i+1][j],
                               vb[i][j-1]);
         num++;
         }
```

```
      if ((i>0)&&(j>0))
         {
         kn += NormalVectorCos(vb[i][j],
                               vb[i-1][j],
                               vb[i][j-1]);
         num++;
         }
      if ((i>0)&&(j<10))
         {
         kn += NormalVectorCos(vb[i][j],
                               vb[i-1][j],
                               vb[i][j+1]);
         num++;
         }
      kr[i][j] = kn/(double)num;
      }
//-------а теперь цикл вывода граней поверхности ----
for (i=0;i<10;i++)
  for (j=0;j<10;j++)
      {
      gran[0] = vb[i][j];
      gran[1] = vb[i+1][j];
      gran[2] = vb[i+1][j+1];
      gran[3] = vb[i][j+1];
      kgran[0] = kr[i][j];
      kgran[1] = kr[i+1][j];
      kgran[2] = kr[i+1][j+1];
      kgran[3] = kr[i][j+1];
      MyPolygonGouraud(gran, kgran, 4, clr);
      }
}
//вычисление координат точки кубической поверхности Безье
void BezierPoint(VERTEX *v,double s, double t)
{
int C[4] = {1, 3, 3, 1};
double k, si[4],ti[4],ssi[4],tti[4];
int i,j;

si[0] = 1;
ti[0] = 1;
ssi[3] = 1;
tti[3] = 1;
```

```
for (i=1;i<4;i++)
    {
    si[i] = s*si[i-1];
    ti[i] = t*ti[i-1];
    ssi[3-i] = (1-s)*ssi[4-i];
    tti[3-i] = (1-t)*tti[4-i];
    }
for (i=0;i<4;i++)
    {
    si[i] = (double)C[i]*si[i]*ssi[i];
    ti[i] = (double)C[i]*ti[i]*tti[i];
    }
v->x = 0;
v->y = 0;
v->z = 0;
for (i=0; i<=3; i++)
    for (j=0; j<=3; j++)
        {
        k = si[i]*ti[j];
        v->x += k*Px[i][j];
        v->y += k*Py[i][j];
        v->z += k*Pz[i][j];
        }
}
//-------определение коэффициентов преобразования ------
//-----------для экранных координат---------------------
void InitViewTranform(double alpha, double beta,
                      int cx, int cy)
{
alpha *= M_PI/180.0;
beta  *= M_PI/180.0;
Aview = cos(alpha);
Bview = -sin(alpha);
Cview = sin(alpha)*cos(beta);
Dview = cos(alpha)*cos(beta);
Eview = -sin(beta);
Fview = sin(alpha)*sin(beta);
Gview = cos(alpha)*sin(beta);
Hview = cos(beta);
Xview = cx/2;
Yview = cy/2;
}
```

```
//------преобразование мировых координат в экранные-----
void ScreenKoord(VERTEX *v)
{
VERTEX tmp;

tmp.x = v->x*Aview + v->y*Bview + Xview;
tmp.y = v->x*Cview + v->y*Dview + v->z*Eview + Yview;
tmp.z = v->x*Fview + v->y*Gview + v->z*Hview;
*v = tmp;
}
//----косинус угла нормали для треугольника----------
double NormalVectorCos(VERTEX v1, VERTEX v2, VERTEX v3)
{
double nx,ny,nz;

v2.x -= v1.x;
v2.y -= v1.y;
v2.z -= v1.z;
v3.x -= v1.x;
v3.y -= v1.y;
v3.z -= v1.z;
nx =  v2.y * v3.z - v3.y * v2.z;
ny = -v2.x * v3.z + v3.x * v2.z;
nz =  v2.x * v3.y - v3.x * v2.y;
return 0.2 + 0.8*fabs(nz)/sqrt(nx*nx+ny*ny+nz*nz);
}
//------------функции графики для DIB растра----------
BOOL InitMyZbuffer(long cx, long cy)
{
pZbuf = new float [cx * cy];
if (pZbuf == NULL) return FALSE;
horSize = cx;
vertSize = cy;
return TRUE;
}
void ClearMyZbuffer(void)
{
long i,size;

if (pZbuf == NULL) return;
size = horSize * vertSize;
for (i=0; i<size; i++)
    pZbuf [i] = -10000;
}
```

```
BOOL OpenRastrMem(long hor,long vert)
{
long bmWidth;

pBitInfo = new BITMAPINFO [sizeof(BITMAPINFO)];
if (pBitInfo == NULL) return FALSE;
bmWidth = hor & 0xfffc;
if (bmWidth < hor) bmWidth += 4;
bmWidth += 8;
bytesPerString = 3*bmWidth;
HorBuf = hor;
VertBuf = vert;
SizeBuf = bytesPerString*VertBuf;
pBitInfo->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
pBitInfo->bmiHeader.biWidth = bmWidth;
pBitInfo->bmiHeader.biHeight = vert;
pBitInfo->bmiHeader.biPlanes = 1;
pBitInfo->bmiHeader.biBitCount = 24;
pBitInfo->bmiHeader.biClrUsed = 0;
pBitInfo->bmiHeader.biClrImportant = 0;
pBitInfo->bmiHeader.biCompression = BI_RGB;
pRastBuf = new BYTE[SizeBuf];
if (pRastBuf == NULL)
   {
   delete []pBitInfo;
   return FALSE;
   }
return TRUE;
}
void CloseRastrMem(void)
{
delete []pRastBuf;
delete []pBitInfo;
}
void ClearRastrMem(void)
{
memset(pRastBuf,255,SizeBuf);
}
void MyPolygonGouraud(VERTEX *v,double *kr,int nv,
                      COLORREF clr)
{
double x[4],z[4],k[4];
int i,y,nhor,min,max,st,en;
double x1,x2,y1,y2,z1,z2,k1,k2;
```

```
min = v[0].y;
max = v[0].y;
for (i=1; i<nv; i++)
    {
    if (min > v[i].y) min = v[i].y;
    if (max < v[i].y) max = v[i].y;
    }
for (y=min; y<=max; y++)
    {
    nhor = 0;
    for (i=0; i<nv; i++)
        {
        st = i;
        en = i+1;
        if (en >= nv) en=0;
        y1 = v[st].y;
        y2 = v[en].y;
        if ((y >= y1)&&(y < y2)||
            (y <= y1)&&(y > y2))
            if (y1 != y2)
               {
               x1 = v[st].x;
               z1 = v[st].z;
               k1 = kr[st];
               x2 = v[en].x;
               z2 = v[en].z;
               k2 = kr[en];
               x[nhor] = x1+(x2-x1)*(double)(y-y1)/
                                    (double)(y2-y1);
               z[nhor] = z1 + (z2-z1)*(double)(y-y1)/
                                      (double)(y2-y1);
               k[nhor] = k1 + (k2-k1)*(double)(y-y1)/
                                      (double)(y2-y1);
               nhor++;
               }
         }
    if (nhor == 2) MyLineHorGouraud(x[0],x[1],y,
                                    z[0],z[1],
                                    k[0],k[1], clr);
    }
}
```

```
//----горизонталь закрашивания полигона методом Гуро-----
void MyLineHorGouraud(int x1,int x2,int y,
                      double z1,double z2,
                      double k1,double k2,COLORREF clr)
{
int x;
double z,k,kz,kk;
BYTE red,gre,blu;

red = GetRValue(clr);
gre = GetGValue(clr);
blu = GetBValue(clr);
if (x1 == x2)
   {
   SetPixMyZ(x1, y, z1, RGB((BYTE)(k1*(double)red),
                            (BYTE)(k1*(double)gre),
                            (BYTE)(k1*(double)blu)));
   return;
   }
kz = (z2-z1)/(double)(x2-x1);
kk = (k2-k1)/(double)(x2-x1);
if (x1 < x2)
   for (x=x1; x<x2; x++)
       {
       z = z1 + (double)(x - x1)*kz;
       k = k1 + (double)(x - x1)*kk;
       SetPixMyZ(x, y, z, RGB((BYTE)(k*(double)red),
                              (BYTE)(k*(double)gre),
                              (BYTE)(k*(double)blu)));
       }
if (x1 > x2)
   for (x=x2; x<x1; x++)
       {
       z = z1 + (double)(x - x1)*kz;
       k = k1 + (double)(x - x1)*kk;
       SetPixMyZ(x, y, z, RGB((BYTE)(k*(double)red),
                              (BYTE)(k*(double)gre),
                              (BYTE)(k*(double)blu)));
       }
}
void SetPixMyZ(int x, int y, float z, COLORREF clr)
{
long indx,lKoord;
```

```
if ((x<0) || (x>=horSize) || (y<0) || (y>=vertSize))
   return;
indx = (long)y*horSize + (long)x;
if (z >= pZbuf [indx])
   {
   pZbuf [indx] = z;
   lKoord = bytesPerString * (VertBuf-y-1) + 3*x;
   pRastBuf[lKoord] = GetBValue(clr);
   pRastBuf[lKoord+1] = GetGValue(clr);
   pRastBuf[lKoord+2] = GetRValue(clr);
   }
}
```

Для упрощения программы поверхности рисуются симметрично по четырем квадрантам.

Всего создается сто кадров. Номер кадра определяет значение параметра t от 0 до 1. Этот параметр используется для расчета текущих координат точки каждой из 16 кубических кривых Безье. Вдоль этих кривых скользят опорные точки-ориентиры поверхностей Безье (рис. 9.1). Каждая поверхность отображается полигональной сеткой 10×10.

Результат работы программы изображен на рис. 9.2.

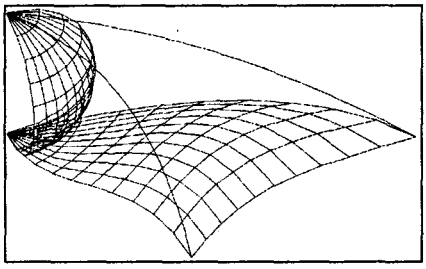


Рис. 9.1. Две трассы движения

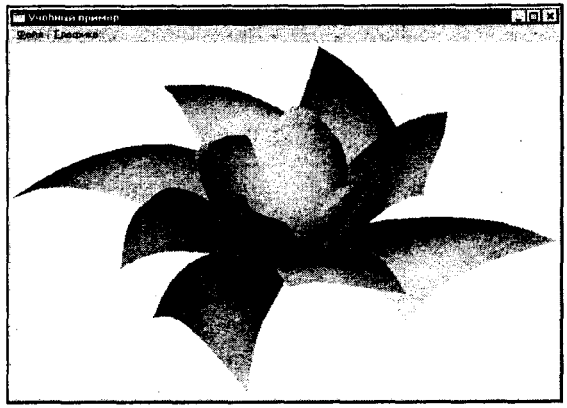


Рис. 9.2. Двенадцать сплайнов Безье. Последний кадр

На рис. 9.3 отображена начальная форма и три промежуточные стадии развития объекта.

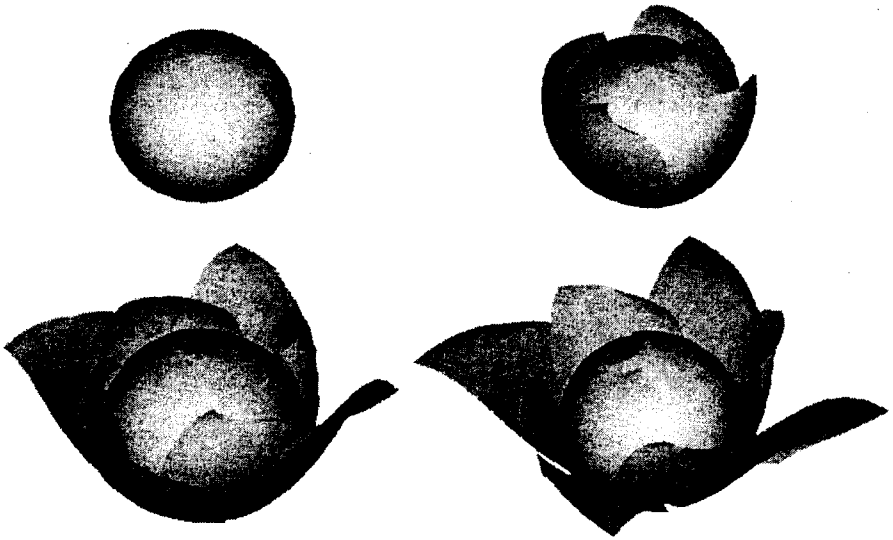


Рис. 9.3. Четыре стадии изменения формы

9.2. Градиентное закрашивание

При разработке и тестировании функций предыдущей программы почти случайно был получен интересный рисунок — пример градиентного закрашивания поверхностей Безье (рис. 9.4 и на обложке книги).

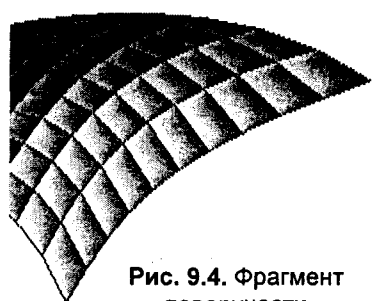


Рис. 9.4. Фрагмент поверхности

На этом рисунке поверхность выглядит текстурированной рельефными ячейками, будто пуховое одеяло. Каждая ячейка — это грань поверхности Безье, которая рисуется так, что цвет закрашивания темнеет слева направо. Для каждой грани вычисляется наклон нормали, задающий цвет градиентного закрашивания. Объект в целом напоминает, скорее всего, капусту.

Текст программы приведем только для тех функций, которые изменены. Поскольку метод Гуро здесь уже не используется, то функция `MyPolygonGouraud` переименована в `MyPolygon`, а `MyLineHorGouraud` — в `MyLineHor`. В теле функции `MyPolygon` вычисляется цвет грани с помощью новой функции `ReflectionColor`. В остальном текст программы идентичен тексту предыдущей программы.

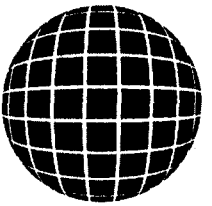
Часть текста программы studex41:

```
//---------Капуста  (c)Copyright Порев В.Н.----------
void DrawBezierSurface(COLORREF clr)
{
int i,j;
VERTEX gran[4];
VERTEX vb[11][11];
VERTEX v;

for (i=0;i<11;i++)
  for (j=0;j<11;j++)
      {
      BezierPoint(&v, (double)i/10, (double)j/10);
      ScreenKoord(&v);

      vb[i][j] = v;
      }
for (i=0;i<10;i++)
  for (j=0;j<10;j++)
      {
      gran[0] = vb[i][j];
      gran[1] = vb[i+1][j];
      gran[2] = vb[i+1][j+1];
      gran[3] = vb[i][j+1];
      MyPolygon(gran, 4, clr);
      }
}

//--рисование полигона грани с градиентным закрашиванием--
void MyPolygon(VERTEX *v,int nv,COLORREF clr)
{
double x[4],z[4];
int i,y,nhor,minY,maxY,st,en;
double x1,x2,y1,y2,z1,z2;

clr = ReflectionColor(clr, v[0], v[1], v[2]);
minY = v[0].y;
maxY = v[0].y;
for (i=1; i<nv; i++)
    {
    if (minY > v[i].y) minY = v[i].y;
    if (maxY < v[i].y) maxY = v[i].y;
    }
```

```
for (y=minY; y<=maxY; y++)
    {
    nhor = 0;
    for (i=0; i<nv; i++)
        {
        st = i;
        en = i+1;
        if (en >= nv) en=0;
        x1 = v[st].x;
        y1 = v[st].y;
        z1 = v[st].z;
        x2 = v[en].x;
        y2 = v[en].y;
        z2 = v[en].z;
        if ((y >= y1)&&(y < y2)||
            (y <= y1)&&(y > y2))
            if (y1 != y2)
               {
               x[nhor] = x1 + (x2-x1)*(double)(y-y1)
                                    /(double)(y2-y1);
               z[nhor] = z1 + (z2-z1)*(double)(y-y1)
                                    /(double)(y2-y1);
               nhor++;
               }
         }
    if (nhor == 2)
       {
       if (x[0] <= x[1])
            MyLineHor(x[0],x[1],y,
                      z[0],z[1],
                      1, 0.5, clr);
       else MyLineHor(x[1],x[0],y,
                      z[1],z[0],
                      1, 0.5, clr);
       }
    }
}

//--------горизонталь закрашивания полигона--------------
void MyLineHor(int x1,int x2,int y,
               double z1,double z2,
               double k1,double k2,COLORREF clr)
```

```
{
int x;
double z,k,kz,kk;
BYTE red,gre,blu;

if (x1 == x2)
   {
   SetPixMyZ(x1, y, z1, 0);
   return;
   }
red = GetRValue(clr);
gre = GetGValue(clr);
blu = GetBValue(clr);
kz = (z2-z1)/(double)(x2-x1);
kk = (k2-k1)/(double)(x2-x1);
if (x1 < x2)
   for (x=x1+1; x<x2; x++)
       {
       z = z1 + (double)(x - x1)*kz;
       k = k1 + (double)(x - x1)*kk;

       SetPixMyZ(x, y, z, RGB((BYTE)(k*(double)red),
                              (BYTE)(k*(double)gre),
                              (BYTE)(k*(double)blu)));

       }
if (x1 > x2)
   for (x=x2+1; x<x1; x++)
       {
       z = z1 + (double)(x - x1)*kz;
       k = k1 + (double)(x - x1)*kk;
       SetPixMyZ(x, y, z, RGB((BYTE)(k*(double)red),
                              (BYTE)(k*(double)gre),
                              (BYTE)(k*(double)blu)));
       }
SetPixMyZ(x1, y, z1, 0);     //концы горизонтали
SetPixMyZ(x2, y, z2, 0);     //черным цветом
}

//--------------вычисление цвета грани----------------
//----------------диффузное отражение-----------------
COLORREF ReflectionColor(COLORREF clr,
                         VERTEX v1, VERTEX v2, VERTEX v3)
```

```
{
double k;

k = NormalVectorCos(v1, v2, v3);
return RGB(k*(double)GetRValue(clr),
           k*(double)GetGValue(clr),
           k*(double)GetBValue(clr));
}
```



ГЛАВА 10

Графическая библиотека OpenGL

Библиотека OpenGL (Open Graphic Library), разработанная фирмой Silicon Graphics, стала индустриальным стандартом. Интерфейс OpenGL поддерживается многими операционными системами для разнообразных аппаратных платформ — от персональных компьютеров до сверхмощных суперкомпьютеров. Важным аспектом является также поддержка интерфейса OpenGL производителями аппаратных графических акселераторов. Поэтому OpenGL позволяет достаточно просто создавать быстродействующие графические программы и часто используется разработчиками компьютерных игр, например, Quake. Библиотека OpenGL поддерживается в операционной системе Windows, начиная с Windows 95 версии OSR 2, — были добавлены соответствующие модули DLL, а также включены несколько функций и структур данных в API Win32.

Интерфейс OpenGL реализован в виде набора функций, которые можно использовать в прикладных программах. Известно также расширение для OpenGL — библиотека классов Open Inventor.

Разработка графических программ OpenGL для среды Windows подобна программированию графики GDI функций API, которое мы рассмотрели в главах 5—8. Однако есть особенности, некоторые из которых мы изучим. Для получения более подробных сведений можно порекомендовать литературные источники — прежде всего, это документация Windows SDK [61]. В значительной мере этот источник был использован в книге [25].

Быстродействие графических программ, использующих OpenGL, существенно зависит от видеоадаптера. Аппаратная реализация всех базовых функций OpenGL — залог высокого быстродействия. В настоящее время многие видеоадаптеры содержат специальный графический процессор (один или несколько) для поддержки функции графики. Кроме того, что видеоадаптер должен аппаратно выполнять все базовые функции OpenGL (такие как преобразования координат, расчеты освещения, наложение текстур, отсечение,

вывод полигонов), для достижения высокого быстродействия должен быть установлен специальный драйвер. Драйверы типа ICD (Installable Client Driver) обеспечивают интерфейс, способствующий эффективному использованию аппаратных возможностей видеоадаптера. Другой тип драйвера — MCD — устанавливается обычно тогда, когда не все функции поддержаны аппаратно, и в этом случае они выполняются программно центральным процессором, что существенно медленнее.

Рассмотрим создание программ OpenGL на языке C, C++ в среде Windows. В главе 6 при рассмотрении графики GDI мы определили ключевой момент — это создание контекста графического устройства (device context). Графика OpenGL в этом плане похожа — необходимо сначала создать контекст, который здесь назван ***контекстом отображения*** (rendering context), и направить текущий вывод графики на него. Потом следует закрыть этот контекст, освободить память.

Будем программировать в стиле программ `StudEx` предыдущих глав данной книги. Этот стиль заключается в непосредственном вызове функций API Windows без каких-либо посредников типа MFC (или иных подобных библиотек). Во-первых, это уменьшает выполняемый код (поскольку каждому посреднику нужно платить — вот только здесь за что?), а во-вторых, позволит нам более детально ознакомиться с OpenGL как таковой. Дадим общую схему программы OpenGL.

1. Создание окна программы. Здесь необходимо обязательно установить стиль окна `WS_CLIPCHILDREN` и `WS_CLIPSIBLINGS`. Это осуществляется заданием значений аргументов функции `CreateWindow`.
2. После создания окна можно открывать контекст отображения. Рекомендуется открытие этого контекста делать во время обработки сообщения `WM_CREATE`.
3. Чтобы создать контекст отображения, сначала необходимо открыть контекст окна (`hdc`), например, функцией `GetDC`.
4. Для выяснения характеристик контекста отображения устанавливаем соответствующие значения полей структуры `PIXELFORMATDESCRIPTOR` и вызываем функцию `ChoosePixelFormat`. Эта функция возвращает номер пиксельного формата, который можно использовать. Если это номер 0, то создание нужного контекста отображения невозможно.
5. Вызовом функции `SetPixelFormat` задаем соответствующий пиксельный формат в контексте `hdc`.
6. На основе контекста `hdc` создаем контекст отображения `hglrc` вызовом функции `wglCreateContext`. Для переадресации текущего вывода графики OpenGL в `hglrc` необходимо вызывать функцию `wglMakeCurrent`.

7. В ходе работы программы выводим графические объекты в текущий контекст отображения. Графический вывод можно осуществлять во время обработки сообщения `WM_PAINT` или других сообщений. Для этого используются функции для работы с графическими примитивами OpenGL.
8. Перед закрытием окна программы необходимо закрыть все открытые контексты отображения. Также следует закрыть все контексты графического устройства. Это можно сделать в ходе обработки сообщения `WM_DESTROY` вызовом функций `ReleaseDC` и `wglDeleteContext`.

Чтобы использовать библиотеку OpenGL, в среде разработки программ на C и C++ необходимо подключить соответствующие файлы заголовков. Например, в среде Borland C++ 5.02 для этого достаточно включить в текст программы строки:

```
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
```

10.1. Пример программы OpenGL

Текст программы здесь представлен в виде двух файлов — `WinOpGL.cpp` и `Studex50.cpp`. В этой программе использованы те же файлы ресурсов (`studex.rc`) и общего описания (`studex.def`), что и во всех предыдущих примерах программ.

Файл `WinOpGL.cpp`:

```
#define STRICT
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <mem.h>
#include <math.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam);
void InitMyOpenGL(HWND hWnd);
void ResizeMyOpenGL(HWND hWnd);
void CloseMyOpenGL(HWND hWnd);
void DrawMyExampleOpenGL(HWND hWnd);
```

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCmd,
                    int nCmdShow)
{
MSG msg;
HWND hWnd;
WNDCLASS WndClass;

WndClass.style = NULL;
WndClass.lpfnWndProc  = WndProc;
WndClass.cbClsExtra   = 0;
WndClass.cbWndExtra   = 0;
WndClass.hInstance    = hInstance;
WndClass.hIcon        = LoadIcon(NULL,IDI_APPLICATION);
WndClass.hCursor      = LoadCursor (NULL, IDC_ARROW);
WndClass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
WndClass.lpszMenuName = "STUDEXMENU";
WndClass.lpszClassName= "StudEx";
if (!RegisterClass(&WndClass)) return 0;
hWnd = CreateWindow("StudEx",
                    "Учебный пример",
                     WS_OVERLAPPEDWINDOW |
                     WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
                     WS_CAPTION | WS_SYSMENU,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     600,
                     450,
                     NULL,
                     NULL,
                     hInstance,
                     NULL);
if (!hWnd) return NULL;
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while (GetMessage(&msg,NULL,NULL,NULL))
  {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
  }
return msg.wParam;
}
```

```
LRESULT CALLBACK WndProc(HWND hWnd,UINT message,
                         WPARAM wParam,LPARAM lParam)
{
switch (message)
  {
  case WM_CREATE:
    InitMyOpenGL(hWnd);
    break;
  case WM_SIZE:
    ResizeMyOpenGL(hWnd);
    break;
  case WM_COMMAND:
    switch (LOWORD(wParam))
      {
      case 201:                    //пункт меню Графика
        DrawMyExampleOpenGL(hWnd);
        break;
      case 108:
        DestroyWindow(hWnd);
        break;
      default:break;
      }
    break;
  case WM_DESTROY:
    CloseMyOpenGL(hWnd);
    PostQuitMessage(0);
    break;
  default:return DefWindowProc(hWnd, message,
                               wParam, lParam);
  }
return 0L;
}

void InitMyOpenGL(HWND hWnd)
{
HDC hdc;
HGLRC hglrc;
PIXELFORMATDESCRIPTOR pfd;
int iPixelFormat;

memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
```

```
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = 24;
pfd. cDepthBits = 32;
pfd. iLayerType = PFD_MAIN_PLANE;
//---теперь создаем контекст отображения OpenGL (hglrc)----
//---по образцу контекста окна (hdc)----------------------
hdc = GetDC(hWnd);
iPixelFormat = ChoosePixelFormat(hdc, &pfd);
SetPixelFormat(hdc, iPixelFormat, &pfd);
hglrc = wglCreateContext(hdc);
if (hglrc) wglMakeCurrent(hdc, hglrc);
}

//-----учитываем возможные изменения размеров окна------
void ResizeMyOpenGL(HWND hWnd)
{
RECT rc;

GetClientRect(hWnd,&rc);
glViewport(0,0,rc.right, rc.bottom);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
}

//--закрытие открытых контекстов: графического устройства
//-----------и контекста отображения OpenGL------------
void CloseMyOpenGL(HWND hWnd)
{
HGLRC hglrc;
HDC hdc;

hglrc = wglGetCurrentContext();
if (hglrc)
   {
   hdc = wglGetCurrentDC();
   wglMakeCurrent(NULL, NULL);
   ReleaseDC (hWnd, hdc);
   wglDeleteContext(hglrc);
   }
}
```

Файл `Studex50.cpp`:

```
//-------------- (c)Copyright Порев В.Н. -----------------
#include "winopgl.cpp"
void DrawMyExampleOpenGL(HWND hWnd)
{
GLubyte fillpattern[128];
gluOrtho2D(-400, 400, -300, 300);
glMatrixMode(GL_MODELVIEW);
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

glColor3f(0, 0, 0);
glLineWidth(2.0f);
glLineStipple(3, 0x08ff);
glEnable(GL_LINE_STIPPLE);
glBegin(GL_LINES);
glVertex2f(-400, 0);
glVertex2f(400, 0);
glEnd();
glBegin(GL_LINES);
glVertex2f(0, 300);
glVertex2f(0,-300);
glEnd();
glDisable(GL_LINE_STIPPLE);

glColor3f(1.0, 0.0, 0.0);
glPointSize(10);
glBegin(GL_POINTS);
glVertex2f(0, 0);
glEnd();

glLineWidth(3.0f);
glBegin(GL_LINE_STRIP);
for (int i=0; i<10; i++)
    {
    glVertex2f(-350+70*i, 60);
    glVertex2f(-315+70*i,-60);
    }
glEnd();

glColor3f(0.0, 1.0, 0.0);
glLineWidth(1.0f);
```

```
glBegin(GL_TRIANGLES);
glVertex2f(-300, 100);
glVertex2f(-200, 290);
glVertex2f(-100, 100);
glEnd();

glBegin(GL_QUADS);
glVertex2f(0, 150);
glVertex2f(200, 200);
glVertex2f(400, 150);
glVertex2f(200, 100);
glEnd();

memset(fillpattern,0,sizeof(fillpattern));
for (int j=0;j<16;j++)
  for (int i=0; i<2; i++)
      {
      fillpattern[j*4+i] = 0xff;
      fillpattern[64+j*4+i+2] = 0xff;
      }
glColor3f(0, 0, 1.0);
glPointSize(1);
glLineWidth(1.0f);
glPolygonStipple(fillpattern);
glEnable(GL_POLYGON_STIPPLE);
glBegin(GL_POLYGON);
glVertex2f(-300, -150);
glVertex2f(-200, -80);
glVertex2f( 0, -100);
glVertex2f( 200, -200);
glVertex2f(-100, -250);
glEnd();
glDisable(GL_POLYGON_STIPPLE);

glFinish();
}
```

Текст этой программы составлен их двух частей — `WinOpGL.cpp` и `Studex50.cpp`. В файле `WinOpGL.cpp` сосредоточены функции, необходимые для создания окна, оконные функции, функции инициализации графики. Этот файл будет использован и в следующих примерах программ OpenGL. Файл `Studex50.cpp` содержит текст, описывающий графическое отображение конкретных объектов (функция `DrawMyExampleOpenGL`). Как вы, наверное, уже за-

метили, все это подобно использованию в главах 5—8 наших собственных файлов `winmain.cpp`, `winmain1.cpp` и `studexXX.cpp`.

Запустите программу, затем выберите меню "Графика". На экране в окне программы появляется картинка, показанная на рис. 10.1.

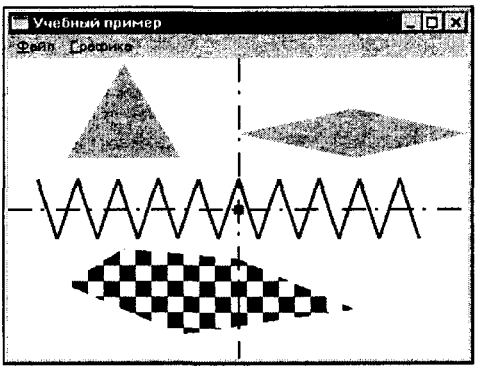


Рис. 10.1. Первый пример OpenGL — двумерная графика

Изображение в окне программы `Studex50` создается из нескольких графических примитивов. В данном случае рисовались точки, линии и полигоны. Вывод каждого такого примитива в OpenGL оформлен парой функций `glBegin` и `glEnd`:

```
glBegin(mode);
. . . .          //здесь перечисляются вершины объектов
glEnd();
```

Аргументом функции `glBegin` является код типа объекта.

Координаты вершины объекта задаются функцией `glVertexXX`. Эта функция имеет много разновидностей (суффиксов `XX`). Отличия обусловлены типом и количеством аргументов `glVertex`. Количество аргументов соответствует числу измерений систем координат. Тип координат-аргументов может быть целым или вещественным (с плавающей точкой) в нескольких разновидностях. Например:

```
glVertex2f(0,300);
```

задает двумерные вещественные координаты, а

```
glVertex3f(0,3,0);
```

задает также вещественные, но трехмерные координаты вершины.

Перечисление всех вершин объекта в программе завершает вызов функции `glEnd`. Это означает запись примитива в очередь графического вывода. В за-

висимости от аргумента функции `glBegin(mode)` список вершин может трактоваться OpenGL по-разному (табл. 10.1).

Таблица 10.1

mode	Как выполняется графический вывод
GL_POINTS	Каждая вершина рисуется как отдельная точка
GL_LINES	Каждая пара вершин соединяется отрезком прямой
GL_LINE_STRIP	Полилиния из нескольких связанных отрезков
GL_LINE_LOOP	Замкнутая полилиния
GL_TRIANGLES	Тройки вершин образуют треугольники
GL_TRIANGLE_STRIP	Связанные треугольники
GL_TRIANGLE_FAN	Связанные треугольники с общей первой вершиной
GL_QUADS	Каждые четыре вершины образуют четырехугольники
GL_QUAD_STRIP	Связанные четырехугольники
GL_POLYGON	Один выпуклый полигон

Можно считать существенным недостатком ограничение для полигонов (`GL_POLYGON`) возможностью вывода только *выпуклых* фигур. Функция API Windows `Polygon` в этом плане намного совершеннее — она рисует и невыпуклые полигоны. В OpenGL для рисования произвольных полигонов предусмотрена триангуляция.

Размер точек можно задать вызовом `glPointSize()`, толщину линий — `glLineWidth()`. Для задания стиля линий используются функции `glLineStipple`, `glEnable` и `glDisable`, например:

```
glLineStipple(3, 0x08ff);
glEnable(GL_LINE_STIPPLE);
. . . .    //здесь используется этот стиль линий
glDisable(GL_LINE_STIPPLE);
```

причем аргументами функции `glLineStipple()` являются количество повторов пикселов и шаблон пунктира.

Стиль заполнения фигур может быть задан растровым образцом в массиве 32×32 бит.

```
GLubyte fillpattern[128];       //массив 32×32 бит

glPolygonStipple(fillpattern);
glEnable(GL_POLYGON_STIPPLE);
. . . . //здесь используется растровый массив заполнения
glDisable(GL_POLYGON_STIPPLE);
```

Обратите внимание, мы уже несколько раз использовали функции `glEnable()` и `glDisable()`. Это многоцелевые функции. Они предназначены для управления многими разнообразными режимами отображения.

10.2. Координаты и матрицы

В OpenGL используются три типа матриц — видовая матрица, матрица проекции и матрица текстуры. Все они имеют размер 4×4 и определяют преобразования координат так, как описано в главе 2 этой книги.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Для задания 16 элементов матрицы можно использовать функции

```
glLoadMatrixd(const GLdouble *m);
glLoadMatrixf(const  GLfloat *m);
```

которые копируют элементы массива `m[ ]` в текущую матрицу.

Для некоторых часто используемых преобразований предусмотрены функции, которые автоматически заполняют значения коэффициентов. Функция `glLoadIdentity()` устанавливает единичную матрицу текущего преобразования:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Следующие функции заполняют матрицы значениями коэффициентов, соответствующими таким преобразованиям:

`glRotated(a,x,y,z)` и `glRotatef(a,x,y,z)` — поворот;

`glScaled(x,y,z)` и `glScalef(x,y,z)` — растяжение/сжатие;

`glTranslated(x,y,z)` и `glTranslatef(x,y,z)` — сдвиг.

Для того чтобы матрица определенного типа стала текущей, следует вызвать функцию `glMatrixMode(mode)`, где значение `mode = GL_MODELVIEW, GL_PROJECTION` или `GL_TEXTURE`. Видовая матрица определяет преобразования мировых координат в координаты проецирования (видовые координаты). Матрица проекции отвечает за преобразование видовых координат проекции в экранные координаты. Матрица текстуры предназначена для наложения проективных текстур.

Для задания проекций отображения предусмотрены несколько функций. Аксонометрическая проекция (здесь называется ортографической) задается функциями `glOrtho( )` или `gluOrtho2D()`. Центральная проекция устанавливается вызовом функции `gluPerspective()`.

Для задания области отсечения графического вывода используется функция `glViewport()`.

10.3. Пример трехмерной графики

Рассмотрим пример 3D-программы OpenGL. Файл `Studex51.cpp`:

```
//-------------- (c)Copyright Порев В.Н. ----------
#include "winopgl.cpp"
void DrawMyExampleOpenGL(HWND hWnd)
{
RECT rc;

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearDepth( 1.0 );
glEnable(GL_DEPTH_TEST);
//----------задаем ракурс показа----------
GetClientRect(hWnd,&rc);
gluPerspective(50, (double)rc.right/rc.bottom, 1, 40);
glMatrixMode(GL_MODELVIEW);  //определяем видовую матрицу
glLoadIdentity();            //сначала единичная матрица
glTranslatef(0, 0, -10);     //сдвиг по оси Z
glRotatef(27,  1,  0, 0);    //поворот вокруг оси X
glRotatef(-19, 0, 1, 0);     //поворот вокруг Y

//---------------- рисуем пирамиду -------
glColor3f(1, 1, 0);                 //желтый цвет
glBegin(GL_TRIANGLES);              //передняя грань
```

```
glVertex3f( 0,   4, 0);
glVertex3f(-2, 0.7, 2);
glVertex3f( 2, 0.7, 2);
glEnd();

glColor3f(0.8, 0.8, 0);        //еще один оттенок желтого
glBegin(GL_TRIANGLES);              //задняя грань
glVertex3f( 0,   4,  0);
glVertex3f(-2, 0.7, -2);
glVertex3f( 2, 0.7, -2);
glEnd();

glColor3f(0.5, 0.5, 0);        //темно-желтый цвет
glBegin(GL_TRIANGLES);              //правая грань
glVertex3f( 0,   4,  0);
glVertex3f( 2, 0.7,  2);
glVertex3f( 2, 0.7, -2);
glEnd();

glBegin(GL_TRIANGLES);              //левая грань
glVertex3f( 0,   4,  0);
glVertex3f(-2, 0.7,  2);
glVertex3f(-2, 0.7, -2);
glEnd();

glColor3f(0.8, 0.8, 0);
glBegin(GL_QUADS);              //основание пирамиды
glVertex3f(-2, 0.7, -2);
glVertex3f( 2, 0.7, -2);
glVertex3f( 2, 0.7,  2);
glVertex3f(-2, 0.7,  2);
glEnd();
//---------шахматное поле----------
for (int j=-5;j<5;j++)
  for (int i=-5;i<5;i++)
      {
      if ((abs(i+j) % 2) == 0)
           glColor3f(1, 0, 0);          //красные и
      else glColor3f(0.8, 0.8, 0.8);    //серые клетки
      glBegin(GL_QUADS);
      glVertex3f(i,   0, j);
      glVertex3f(i+1, 0, j);
      glVertex3f(i+1, 0, j+1);
```

```
      glVertex3f(i,   0, j+1);
      glEnd();
      }
glFinish();
}
```

Результат работы программы показан на рис. 10.2.

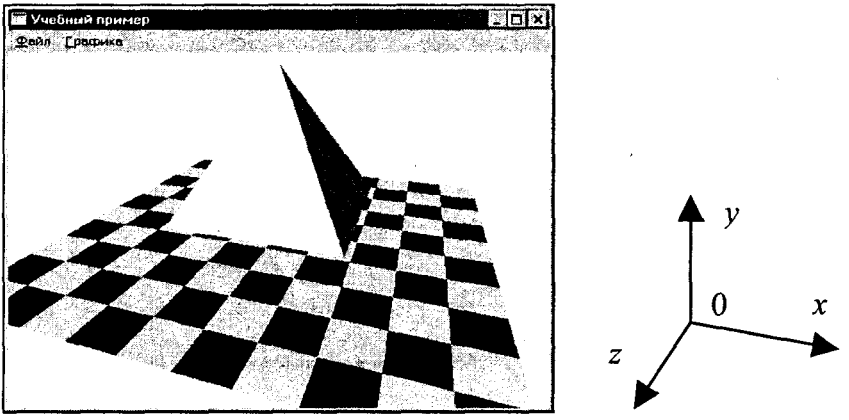


Рис. 10.2. Изображение объектов и направление осей мировых координат в программе `Studex51`

Для иллюстрации работы Z-буфера (в OpenGL он называется буфером глубины — *depth buffer*) сначала рисуем грани пирамиды, а потом шахматное поле, которое расположено ниже. Грани пирамиды также рисуем все пять (хотя здесь видны только две), причем так, чтобы при отсутствии Z-буфера получался бы заведомо неправильный результат. Проверьте это, попробуйте отключить Z-буфер. Для этого достаточно удалить из текста оператор `glEnable(GL_DEPTH_TEST)`. Что получится?

Программа `Studex51` может служить простейшим примером построения изображения трехмерных объектов в заданной проекции. Изображение на рис. 10.2 вполне правдоподобно передает форму объектов. Однако в данной программе отсутствуют многие важные элементы построения реалистичных изображений. В первую очередь это относится к освещению. Так, например, грани пирамиды здесь закрашены разными цветами — передняя грань более светлым цветом, а боковая грань более темным. Эти цвета мы задали вызовом функций `glColor3f` перед выводом соответствующих граней. Значения цветов выбраны произвольно. Здесь никак не были использованы возможности, которые предоставляет OpenGL для моделирования освещения.

10.4. Моделирование освещения

Для того чтобы поручить OpenGL изображать объекты в соответствии с некоторой моделью освещения, необходимо вызвать функцию `glEnable(GL_LIGHTING)`. Может быть использовано несколько источников света. Максимальное количество источников света зависит от версии реализации OpenGL. Источники света имеют номера от 0 до некоторого максимального значения. Включение *i*-го источника выполняется функцией `glEnable(GL_LIGHTi)`, а выключение — функцией `glDisable(GL_LIGHTi)`. Так, например, чтобы заставить светить нулевой источник, необходимо вызвать функцию `glEnable(GL_LIGHT0)`.

Каждый источник света можно (и нужно) настроить индивидуально. Делается это вызовом функций `glLightf`, `glLighti` или `glLightfv`, `glLightiv` (векторные разновидности). Функции различаются типами и количеством аргументов. Рассмотрим некоторые примеры настройки.

Определить положение источника света `GL_LIGHT0` можно следующим образом:

```
GLfloat lightpos[4];
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);
```

В массив `lightpos[]` необходимо записать однородные мировые координаты источника света в виде (*x*, *y*, *z*, *w*). При вызове `glLightfv` эти координаты будут преобразованы в соответствии с ракурсом показа, определяемом видовой матрицей. При *w*=1 будет обеспечено правильное соответствие закрашивания объектов расположению источника.

Направление действия источника света задается так:

```
GLfloat lightdirection[3];
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,lightdirection);
```

В массиве `lightdirection[]` должны быть указаны координаты радиус-вектора в относительных единицах (*x*, *y*, *z*), направленного от источника света.

Угол распространения света от точечного направленного источника можно задать так:

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF, angle);
```

где `angle` — угол в градусах в диапазоне от 0 до 90°. При этом свет в пространстве ограничивается конусом. Также допустимо значение `angle` = 180, которое соответствует равномерному распространению света во все стороны.

Этим далеко не исчерпываются возможности функций `glLightf`, `glLighti`, `glLightfv` и `glLightiv` для установки параметров источников света. Для наиболее полного ознакомления обратитесь к документации по Win32 SDK [61].

Необходимо учитывать, что многие параметры, в том числе и для источников света, в OpenGL установлены по умолчанию. Так, например, для источника GL_LIGHT0 установлен по умолчанию равномерный рассеянный свет (angle = 180). Кроме того, установки по умолчанию неодинаковы для различных источников GL_LIGHTi.

Кроме параметров настройки источника света еще одним важнейшим аспектом моделирования освещения является задание нормалей к поверхностям. При выводе каждого объекта OpenGL определяет взаимную ориентацию вектора направления источника света и текущего вектора нормали. По умолчанию установлено направление вектора нормали как (0, 0, 1), то есть вектор направлен вдоль оси *z* мировых координат. Направление текущего вектора нормали не изменяется до тех пор, пока не будет вызвана функция glNormal, например:

```
glNormal3f(nx,ny,nz);
```

где nx, ny и nz — это координаты радиус-вектора нормали. Для данной функции они должны быть в диапазоне (–1.0, 1.0). Существуют и другие разновидности для glNormal, отличающиеся типами числовых значений координат.

Таким образом, при выводе каждой отдельной полигональной грани необходимо вначале устанавливать направление текущего вектора нормали. Для корректного расчета взаимного расположения векторов нормалей и направлений источников света при различных ракурсах показа необходимо установить режим нормализации векторов

```
glEnable(GL_NORMALIZE);
```

Для правильного вывода освещаемых объектов нужно также определять свойства материала поверхности. Если предполагается изображать освещенными цветные объекты, то следует установить

```
glEnable(GL_COLOR_MATERIAL);
```

Суммируем полученные сведения в программе Studex52.

```
//----------(c) Copyright Порев В.Н. -----------
#include "winopgl.cpp"
//----вычисление координат вектора нормали------
//-------этой функции нет в OpenGL--------------
void NormalVector(float *x,float *y,float *z,
                  float x1,float y1,float z1,
                  float x2,float y2,float z2,
                  float x3,float y3,float z3)
```

```
{
*x = (y2-y1)*(z3-z1) - (y3-y1)*(z2-z1);
*y =-(x2-x1)*(z3-z1) - (x3-x1)*(z2-z1);
*z = (x2-x1)*(y3-y1) - (x3-x1)*(y2-y1);
}
void DrawMyExampleOpenGL(HWND hWnd)
{
RECT rc;
float nx,ny,nz;
GLfloat lightpos[4] = {3,3,4,1};
GLfloat lightdirection[3] = {-0.5, -0.6, -0.7};
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearDepth( 1.0 );
glEnable(GL_DEPTH_TEST);
//----------задаем ракурс показа-----------
GetClientRect(hWnd,&rc);
gluPerspective(50, (double)rc.right/rc.bottom, 1, 40);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -10);
glRotatef(27,  1,  0, 0);
glRotatef(-19, 0, 1, 0);
//--------задаем параметры источника света--------------
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,lightdirection);
glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,4);
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,50);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_NORMALIZE);
//----здесь рисуем только две видимые грани пирамиды------
glColor3f(1, 1, 0);          //желтый цвет пирамиды
NormalVector(&nx, &ny, &nz,  //вектор нормали правой грани
               0,   4,   0,
               2, 0.7,   2,
               2, 0.7,  -2);
glNormal3f(nx, ny, nz);
glBegin(GL_TRIANGLES);
glVertex3f(0,   4,  0);
glVertex3f(2, 0.7,  2);
glVertex3f(2, 0.7, -2);
glEnd();
```

```
NormalVector(&nx, &ny, &nz,     //передняя грань
              0,   4,   0,
             -2, 0.7,   2,
              2, 0.7,   2);
glNormal3f(nx, ny, nz);
glBegin(GL_TRIANGLES);
glVertex3f( 0,   4,  0);
glVertex3f(-2, 0.7,  2);
glVertex3f( 2, 0.7,  2);
glEnd();
//---------------рисуем шахматное поле-----------------
glNormal3f(0,1,0);            //нормаль направлена вверх
for (int j=-5;j<5;j++)
  for (int i=-5;i<5;i++)
      {
      if ((abs(i+j) % 2) == 0)
           glColor3f(1, 0, 0);        //красные и
      else glColor3f(1, 1, 1);        //белые клетки
      glBegin(GL_QUADS);
      glVertex3f(i,   0, j);
      glVertex3f(i+1, 0, j);
      glVertex3f(i+1, 0, j+1);
      glVertex3f(i,   0, j+1);
      glEnd();
      }
glFinish();
}
```

Результат работы программы `Studex52` показан на рис. 10.3.

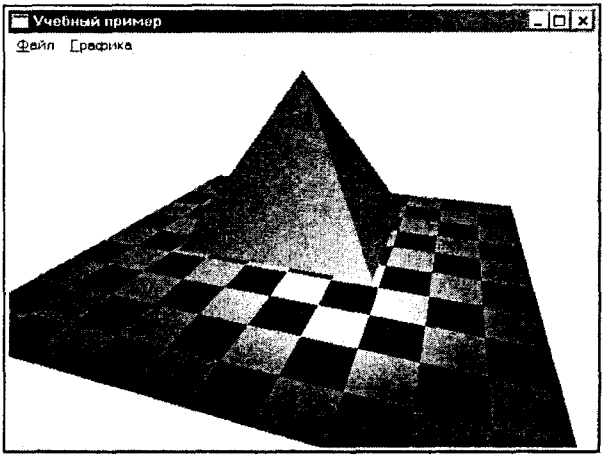


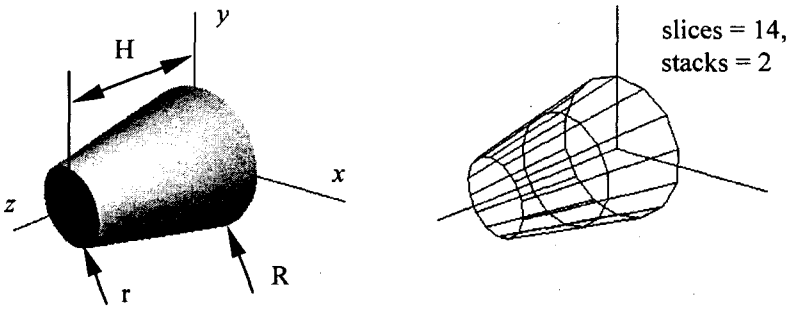
Рис. 10.3. Моделирование освещения (свет справа сверху)

10.5. Стандартные объемные формы

В OpenGL предусмотрены некоторые стандартные, наиболее часто используемые трехмерные объекты. Набор таких форм представлен в библиотеке GLU (Utility Library), которая реализована в виде модуля `glu32.dll` и является неотъемлемой частью OpenGL. Она включает в себя несколько функций управления проекциями (одну из которых — `gluPerspective` мы уже использовали), функции работы с полигонами, кривыми и поверхностями типа B-сплайнов и другие функции.

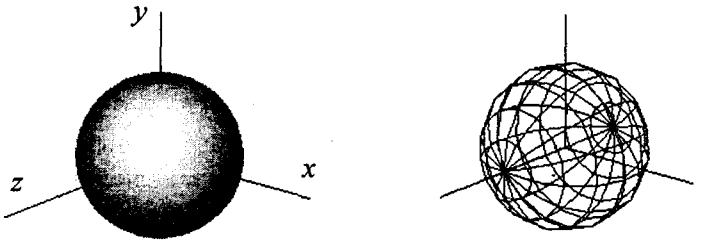
Рассмотрим функции `gluCylinder`, `gluSphere`, `gluDisk` и `gluPartialDisk` (рис. 10.4—10.7).

Перечисленные объекты названы *"quadric objects"*. Параметры `slices` и `stacks` определяют количество плоских граней, используемых для аппроксимации поверхности. Для того чтобы нарисовать подобный объект, необхо-



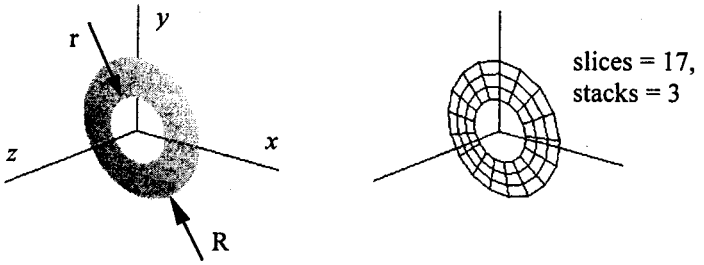

```
gluCylinder(quadricObj, R, r, H, slices, stacks)
```

Рис. 10.4. Цилиндр



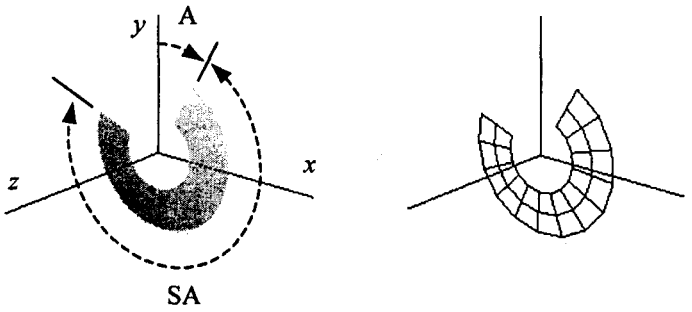

```
gluSphere (quadricObj, R, slices, stacks)
```

Рис. 10.5. Шар




```
gluDisk(quadricObj, r, R, slices, stacks)
```

Рис. 10.6. Диск




```
gluPartialDisk(quadricObj, r, R, slices, stacks, A, SA)
```

Рис. 10.7. Часть диска

димо вызвать функцию `gluNewQuadric`, а после рисования освободить память вызовом функции `gluDeleteQuadric`.

```
quadricObj = gluNewQuadric();
. . . . //рисуем объект quadricObj
gluDeleteQuadric (quadricObj);
```

По умолчанию каждый объект рисуется со сплошным заполнением. Изменить стиль показа можно вызовом функции `gluQuadricDrawStyle`. Можно задать такие стили показа — в виде точек, расположенных в вершинах многоугольника, каркасное изображение, сплошное заполнение и силуэт (разновидность каркасного). Например, вызов

```
gluQuadricDrawStyle (quadricObj, GLU_LINE);
```

дает каркасное изображение.

Объекты данного типа располагаются в пространстве в центре координат (0, 0, 0) с учетом матрицы GL_MODELVIEW. Поэтому, чтобы нарисовать изображение объекта в требуемом месте, нужно соответствующим образом изменить эту матрицу, например, с помощью функций glTranslate и glRotate. Нижеприведенный пример иллюстрирует показ объектов типа *"quadric objects"*.

Файл Studex53.cpp:

```
//----------(c) Copyright Порев В.Н. ----------
#include "winopgl.cpp"
void DrawRocket(GLUquadricObj *quadricObj);
void DrawSaturn(GLUquadricObj *quadricObj);

void DrawMyExampleOpenGL(HWND hWnd)
{
RECT rc;
GLfloat lightpos[4] = {3,3,4,1};
GLfloat lightdirection[3] = {-0.5, -0.6, -0.7};
GLUquadricObj *quadricObj;

glClearColor(0, 0, 0.5, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearDepth( 1.0 );
glEnable(GL_DEPTH_TEST);
//----------задаем ракурс показа-----------
GetClientRect(hWnd,&rc);
gluPerspective(50, (double)rc.right/rc.bottom, 1, 40);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -5);
//--------параметры источника света--------------
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,lightdirection);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_NORMALIZE);
//-----будем рисовать цилиндры, сферу, диск-----
quadricObj = gluNewQuadric();
if (quadricObj)
   {
   glPushMatrix();            //сохраняем матрицу MODELVIEW
   glRotatef(50, 0, 0, 1);    //задаем ориентацию
   glRotatef(120, 0, 1, 0);
```

```
    glTranslatef(0, 1, -2);
    DrawRocket(quadricObj);    //рисуем ракету
    glPopMatrix();       //восстанавливаем матрицу MODELVIEW

    glPushMatrix();
    glTranslatef(2, -1, -3);
    DrawSaturn(quadricObj);     //Сатурн
    glPopMatrix();

    gluDeleteQuadric (quadricObj);
    }
glFinish();
}
//-----ракета изображается несколькими цилиндрами-------
void DrawRocket(GLUquadricObj *quadricObj)
{
glColor3f(1, 0.5, 0);
glTranslatef(0, 0, 2);
gluCylinder (quadricObj, 0.3, 0, 1.5, 16, 1);     //нос
glColor3f(0.8, 0.8, 0.8);
glTranslatef(0, 0, -2.2);
gluCylinder (quadricObj, 0.2, 0.1, 0.2, 16, 1);  //сопло
glTranslatef(0, 0, 0.2);
gluCylinder (quadricObj, 0.3, 0.3, 2, 16, 1);     //корпус
glTranslatef(0.6, 0, 0);
glRotatef(-12,  0,  1, 0);
gluCylinder (quadricObj, 0.3, 0, 1.5, 16, 1);  //двигатели
glRotatef(12,  0,  1, 0);
glTranslatef(-1.2, 0, 0);
glRotatef(12,  0,  1, 0);
gluCylinder (quadricObj, 0.3, 0, 1.5, 16, 1);
glRotatef(-12,  0,  1, 0);
glTranslatef(0.6, 0, 0);
glTranslatef(0, 0.6, 0);
glRotatef(12,  1,  0, 0);
gluCylinder (quadricObj, 0.3, 0, 1.5, 16, 1);
glRotatef(-12,  1,  0, 0);
glTranslatef(0, -1.2, 0);
glRotatef(-12,  1,  0, 0);
gluCylinder (quadricObj, 0.3, 0, 1.5, 16, 1);
}
void DrawSaturn(GLUquadricObj *quadricObj)
{
glColor3f(0, 1, 1);
```

```
gluSphere (quadricObj, 1, 32, 32);
glRotatef(-90,  1,  0, 0);
glRotatef(-15,  0,  1, 0);
gluDisk (quadricObj, 1.5, 2.5, 32, 16);
}
```

Изображение в окне программы `Studex53` приведено на рис. 10.8.

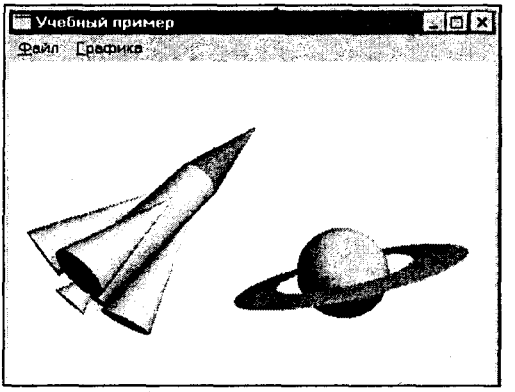


Рис. 10.8. Цилиндры, шар и диск

При подготовке этого рисунка к печати был изменен цвет фона на белый. Однако на экране монитора синий цвет выглядит значительно лучше (в тексте программы: `glClearColor(0, 0, 0.5, 1.0f)`).

10.6. Текстура

В OpenGL имеется достаточно полный набор средств для построения текстурированных изображений.

В качестве текстуры можно использовать растровое изображение. Оно может быть одномерным или двумерным. Для наложения текстуры необходимо выполнить несколько операций.

1. Сначала открыть в памяти массив, в котором будет храниться растр текстуры. Число байт массива рассчитывается исходя из количества бит на пиксел текстуры. Размеры растра текстуры обязательно ***должны быть равны степени двойки*** (плюс несколько пикселов на бордюр). Это требование создает некоторые неудобства для программиста, особенно в случае, когда текстура может быть прочитана из произвольного растрового файла. Впрочем, это не является неразрешимой проблемой — любой растр текстуры можно либо обрезать, либо растянуть (сжать) до требуемых размеров.

2. Заполнить массив текстуры. Здесь следует учитывать то, в каком формате представлен растр текстуры. Если пикселы текстуры представляются в формате RGB (24 бита на пиксел), то байты в массиве должны располагаться в виде троек (R, G, B). Заметим, что в массивах DIB Windows API цветовые компоненты располагаются в обратном порядке, то есть (B, G, R).
3. После того как массив открыт, нужно передать OpenGL адрес массива и другие его параметры. Делается это вызовом функции `glTexImage2D` для двумерной текстуры и `glTexImage1D` для одномерной
4. Затем можно задать параметры фильтрации текстуры (вызовом функции `glTexParameter`) для качественного отображения объектов различных размеров.
5. Перед непосредственным рисованием объектов необходимо установить режим использования текстуры. Делается это вызовом функции `glEnable(GL_TEXTURE_2D)`. Для объектов типа *"quadric objects"* (шар, цилиндр, диск) нужно также вызвать еще и функцию `gluQuadricTexture(quadricObj, GL_TRUE)`.
6. При выводе полигональных граней (`GL_TRIANGLES`, `GL_QUADS` и им подобных) необходимо указывать соответствие текстурных координат и координат в пространстве объектов. Сделать это можно вызовом функций из семейства `glTexCoord`. Так, например, функция `glTexCoord2f(s,t)` указывает на точку с текстурными координатами *s* и *t*. Последующий вызов функции `glVertex3f(x,y,z)` одновременно с заданием координат грани также ставит в соответствие координаты (*s,t*) координатам (*x,y,z*). На рис. 10.9 показан пример отображения координат четырехугольной грани.

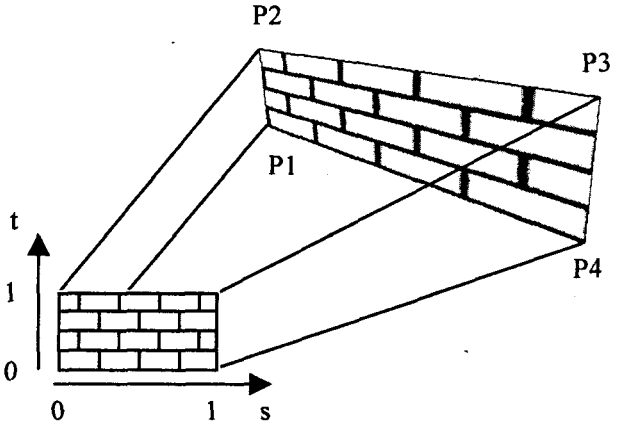


Рис. 10.9. Координаты текстуры (*s, t*) и координаты вершин (*P1—P4*)

Запрограммировать такое отображение текстуры можно таким образом:

```
glTexCoord2f(0, 0); glVertex3f(P1x, P1y, P1z);
glTexCoord2f(0, 1); glVertex3f(P2x, P2y, P2z);
glTexCoord2f(1, 1); glVertex3f(P3x, P3y, P3z);
glTexCoord2f(1, 0); glVertex3f(P4x, P4y, P4z);
```

Рассмотрим пример использования текстуры в следующей программе.

Файл `Studex54.cpp`:

```
//----------(c) Copyright Порев В.Н.------------
#include "winopgl.cpp"
#define horTexture 64
#define vertTexture 32
BYTE pixels[horTexture*vertTexture*3]; //24 бит/пиксел

void InitMyTexture(void);

void DrawMyExampleOpenGL(HWND hWnd)
{
RECT rc;
GLfloat lightpos[4] = {9,4,15,1};
GLfloat vX[5] = {2, 2,-2,-2, 2};  //координаты стен
GLfloat vYZ[5] = {2,-2,-2, 2, 2};
GLfloat nX[4]  = {1,0,1,0};        //нормали
GLfloat nZ[4]  = {0,1,0,1};
GLUquadricObj *quadricObj;

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearDepth( 1.0 );
glEnable(GL_DEPTH_TEST);
//----------задаем ракурс показа------------------
GetClientRect(hWnd,&rc);
gluPerspective(50, (double)rc.right/rc.bottom, 1, 40);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -6);
glRotatef(21,  1,  0, 0);
glRotatef(-40, 0, 1, 0);
//--------задаем параметры источника света---------
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);
glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,1);
```

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,180);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_NORMALIZE);
//---------устанавливаем отображение текстуры------
InitMyTexture();
glEnable(GL_TEXTURE_2D);
//-------------рисуем элементы сцены---------------
glColor3f(0.8, 0.8, 0.8);
for (int i=0;i<4;i++)                    //4 стены
  {
  glNormal3f(nX[i], 0, nZ[i]);
  glBegin(GL_QUADS);
  glTexCoord2f(0, 0); glVertex3f(vX[i],   0, vYZ[i]);
  glTexCoord2f(1, 0); glVertex3f(vX[i+1], 0, vYZ[i+1]);
  glTexCoord2f(1, 1); glVertex3f(vX[i+1], 1, vYZ[i+1]);
  glTexCoord2f(0, 1); glVertex3f(vX[i],   1, vYZ[i]);
  glEnd();
  }
quadricObj = gluNewQuadric();
if (quadricObj)
   {
   for (int i=0;i<4;i++)                 //башни
     {
     glPushMatrix();
     glRotatef(-90,  1,  0, 0);
     glTranslatef(vX[i], vYZ[i], 0);
     glColor3f(0.8, 0.8, 0.8);          //цвет стен башни
     glEnable(GL_TEXTURE_2D);
     gluQuadricTexture(quadricObj, GL_TRUE);
     gluCylinder(quadricObj, 0.4, 0.35, 1.2, 16, 1);
     gluQuadricTexture(quadricObj, GL_FALSE);
     glDisable(GL_TEXTURE_2D);          //крыша без текстуры
     glTranslatef(0, 0, 1.2);
     glColor3f(1, 0.5, 0);              //цвет крыши
     gluCylinder(quadricObj, 0.38, 0, 0.7, 16, 1);
     glPopMatrix();
     }
   gluDeleteQuadric (quadricObj);
   }
glFinish();
}
```

```
//------заполняем массив "шахматной" текстуры--------
void InitMyTexture(void)
{
long krd;
for (long j=0;j<vertTexture;j++)
  for (long i=0;i<horTexture;i++)
    {
    krd = 3*horTexture*j + 3*i;
    pixels[krd]  =255;
    pixels[krd+1]=255;
    pixels[krd+2]=255;
    if ((i/8 + j/8) % 2 == 0)    //голубые клетки
       {
       pixels[krd]=0;            //R
       pixels[krd+1]=128;        //G
       }
    }
glTexImage2D(GL_TEXTURE_2D, 0, 3,
             horTexture,vertTexture,
             0, GL_RGB, GL_UNSIGNED_BYTE,
             pixels);
glTexParameterf(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
}
```

Результат работы программы `Studex54` показан на рис. 10.10.

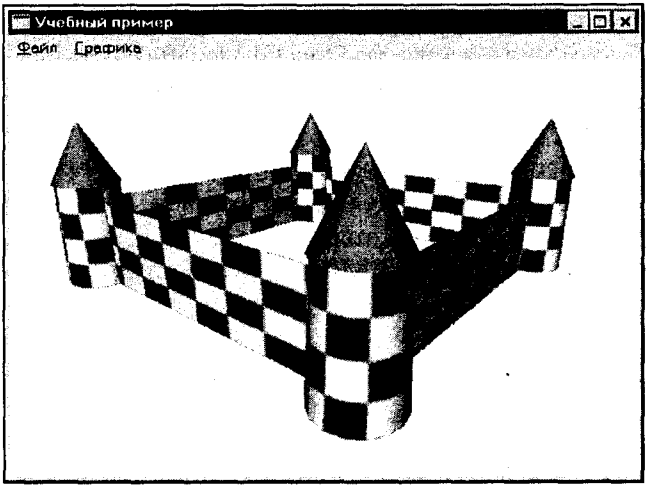


Рис. 10.10. Пример синтетической текстуры

Можно предположить, что хозяин этого замка решил покрыть плиткой стены, дабы уберечь свою недвижимость от разрушительного воздействия агрессивной окружающей среды. Так это было или нет, но здесь мы еще раз использовали шахматный узор — уже для текстуры. Растр текстуры генерируется здесь "на лету" и сохраняется в массиве `pixels`. Это пример синтетической текстуры, узор создается простейшим алгоритмом. Для создания реалистичных изображений в качестве текстур обычно используются цифровые фотографии.

В общем случае, текстуры удобнее хранить в файлах на диске. Это могут быть достаточно сложные изображения, изготовленные заблаговременно. В программах для Windows растры можно создавать в виде ресурсов, которые после компиляции записываются в выполняемые файлы, например, в файлы EXE. А можно использовать и отдельные файлы стандартных форматов. В последнем случае текстуры удобно многократно редактировать.

Рассмотрим, как можно использовать текстуры, записанные в файлах BMP. Такие файлы хранят растр в формате DIB (Device Independent Bitmap). Формат DIB похож на формат текстуры OpenGL, однако есть некоторые отличия. Так, в DIB используется выравнивание строк на границу двойного слова. Иными словами, количество байт в строке растра всегда должно быть кратно четырем — если это не так, то добавляют лишние байты. В нашем случае благоприятным фактором является то, что размеры текстур OpenGL должны быть равны степени двойки. Начиная с размеров по горизонтали, равных четырем, 24-битные растры DIB автоматически располагаются в памяти так же, как и текстуры OpenGL — выравнивание отсутствует.

Если использовать 24-битную глубину цвета, то более существенным отличием DIB от формата текстур OpenGL является порядок расположения байтов R, G и B. Для массивов текстур OpenGL должно быть R-G-B, в то время как в DIB наоборот: B-G-R. Поэтому после чтения файла необходимо переставлять байты R и B.

Наша следующая программа (`Studex55`) иллюстрирует чтение текстуры из файла BMP. Эта программа является модификацией предыдущей программы (`Studex54`). Изменения коснулись только функции `InitMyTexture`. В ее тело встроена функция чтения файлов BMP, которая названа `ReadTextureBMP`.

```
void InitMyTexture(void)
{
ReadTextureBMP("filename.bmp", pixels,
                horTexture, vertTexture);
glTexImage2D(GL_TEXTURE_2D, 0, 3,
             horTexture,vertTexture,
             0, GL_RGB, GL_UNSIGNED_BYTE, pixels);
```

```
glTexParameterf(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
}

BOOL ReadTextureBMP(char *filename,BYTE *lp,
                    DWORD hor,DWORD vert)
{
BITMAPFILEHEADER bfhdr;
BITMAPINFOHEADER bmihdr;
FILE *fin;
DWORD i,j,pos,byteshor;
BYTE bt;

fin = fopen(filename,"rb");
if (fin == NULL) return FALSE;
fread(&bfhdr,sizeof(BITMAPFILEHEADER),1,fin);
fread(&bmihdr,sizeof(BITMAPINFOHEADER),1,fin);
if ((bmihdr.biBitCount != 24)||  //только 24-битные растры,
    (bmihdr.biWidth != hor)||    //соответствующие размерам
    (bmihdr.biHeight != vert))   //массива текстуры
    {
    fclose(fin);                 //этот файл не подходит
    return FALSE;
    }
byteshor = 3*bmihdr.biWidth;   //число байт в строке растра
fseek(fin, bfhdr.bfOffBits, SEEK_SET);
pos = 0;
for (i=0; i<bmihdr.biHeight; i++)
    {
    fread(lp+pos, byteshor, 1,fin);  //читаем строку растра
    pos += byteshor;
    }
fclose(fin);
for (j=0;j<vertTexture;j++)   //меняем местами байты R и B
  {
  pos = 3*horTexture*j;
  for (i=0;i<horTexture;i++)
    {
    bt = lp[pos];
    lp[pos]=lp[pos+2];
    lp[pos+2]=bt;
```

```
    pos += 3;
    }
  }
return TRUE;
}
```

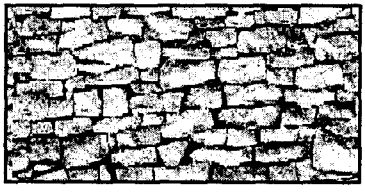
Текстура здесь читается из файла "filename.bmp".

В этой программе текстура загружается в массив всякий раз при создании кадра. В данном примере это вполне допустимо, однако в других случаях такой подход может быть плохим по быстродействию. Поскольку дисковые операции являются медленными (особенно при чтении нескольких различных файлов больших размеров), то чтение файлов текстур нужно стараться делать как можно реже. Например, при построении кадров "облета замка" — то есть при показе одних и тех же объектов с разных ракурсов, текстуры лучше всего загружать перед началом цикла показа. Если используются несколько текстур, то для каждой можно создать в памяти отдельный массив.

На рис. 10. 11 показан результат работы Studex55.

Возможно, в этом замке мало дверей и окон. Но их несложно добавить в текстуру с помощью любого растрового графического редактора, не так ли? Хотя, вероятно, понадобится использовать уже несколько текстур для разных стен.

Следует отметить, что приведенная выше функция ReadTextureBMP не является универсальной — она не рассчитана на другие разновидности формата файлов BMP. Эту функцию необходимо существенно видоизменить, если предусматривается чтение, например, и 256-цветных растров. Такие растры читать несколько сложнее, поскольку требуется загружать и устанавливать палитру. В качестве универсального решения для чтения файлов формата BMP можно порекомендовать использовать функцию auxDIBImageLoad из библиотеки GLAUX.



текстура

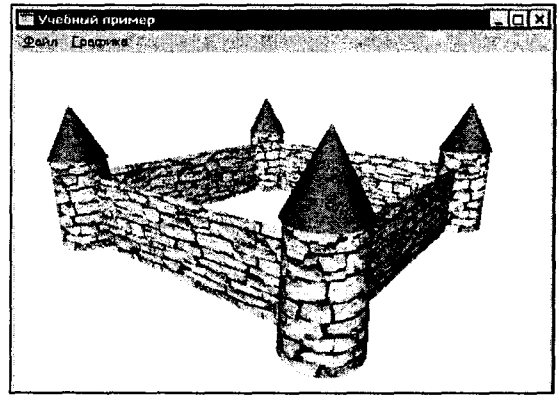


Рис. 10.11. Использование текстуры из файла

Приложение

Здесь приведен исходный текст основных модулей учебной программы для демонстрации метода трассировки лучей.

В модуле SCENE поддерживаются структуры данных для описания сцены и функции создания объектов сцены. Каждый объект представляется одной или несколькими плоскими треугольными или четырехугольными гранями. Параметры материала объекта — собственный цвет поверхности и параметры отражения и пропускания света, такие как фоновое рассеяние (ambient), диффузное (diffuse), зеркальное отражение (reflect), зеркальные блики (specular), гладкость (показатель Фонга p) и прозрачность (through). Описание объектов записывается в три одномерных вектора, поддерживающих иерархию: объекты — грани — вершины. Для ускорения трассировки в структуры данных также записываются оболочки. Есть два типа оболочек — оболочки объектов и оболочки отдельных граней. Оболочки имеют форму шара.

Файл scene.h:

```
//--------(c) Copyright Порев В.Н.----------
#ifndef _SCENE_H_
#define _SCENE_H_
//-------структуры данных------------
struct VERTEX
{
double x,y,z;          //координаты точки в пространстве
};
struct OBJECTPARAM     //параметры объекта
{
long start;            //номер первой грани объекта
long ng;               //число граней
double ambient;        //параметры отражения света
```

```
double diffuse;
double reflect;
double specular;
double p;              //показатель Фонга
double through;        //прозрачность
double r,g,b;          //цвет
double radius;         //радиус оболочки объекта
double xo,yo,zo;       //центр оболочки
};
struct GRANPARAM       //параметры грани
{
long start;            //индекс первой вершины грани
long nv;               //число вершин
double radius;         //радиус оболочки грани
double xo,yo,zo;       //центр оболочки
};
struct LIGHTSRCPARAM //параметры точечного источника света
{
VERTEX pos;            //расположение
double r,g,b;          //цвет
};
extern long ObjectsNum;     //количество объектов
extern long NumGran;        //граней
extern long NumVertex;      //вершин
extern int  NumLightSrc;    //источников света
extern OBJECTPARAM *objarray;     //массив объектов
extern GRANPARAM *grandef;        //массив граней
extern VERTEX *vg;                //массив вершин
extern LIGHTSRCPARAM *lightsrc;   //источники света
extern VERTEX camerapos,          //положение камеры
              cameradir;          //направление обзора
extern double zProjectionPlane;   //положение плоскости проецирования
                                  //на оси Z видовых координат
extern double SpaceColorR,        //цвет свободного пространства
              SpaceColorG,
              SpaceColorB;
extern int  OpenArrays_SCE(long no,long ng,long nv,int nl);
extern void CloseArrays_SCE(void);
extern void SetColor_SCE(double r,double g,double b);
extern void SetMaterial_SCE(double ambient,
                            double diffuse,
                            double reflect,
                            double specular,
                            double p,
                            double through);
```

```
extern void SetCameraPosition_SCE(double x,double y,double z,
                                  double dirx,double diry,
                                  double dirz,double zprojplane);
extern int AddLightSource_SCE(double r,double g,double b,
                              double x,double y,double z);
extern int AddTriangleGran_SCE(double x1,double y1,double z1,
                               double x2,double y2,double z2,
                               double x3,double y3,double z3,
                               int obj);
extern int AddQuadGran_SCE(double x1,double y1,double z1,
                           double x2,double y2,double z2,
                           double x3,double y3,double z3,
                           double x4,double y4,double z4,
                           int obj);
extern int AddPyramid_SCE(double x0,double y0,double z0,
                          double x1,double y1,double z1,
                          double x2,double y2,double z2,
                          double x3,double y3,double z3,
                          double x4,double y4,double z4);
extern int AddPrisma4_SCE(double x1,double y1,
                          double x2,double y2,
                          double x3,double y3,
                          double x4,double y4,
                          double zottom,double ztop);
extern int AddSphere_SCE(double x,double y,double z,
                         double r,int db,int dl);
extern int AddCylinder_SCE(double x,double y,double z,
                           double r,double h,int dl);
extern int CopyObjectAndShift_SCE(double sx,double sy,double sz);
#endif
```

Файл scene.cpp :

```
//--------(c) Copyright Порев В.Н.----------
#include <math.h>
#include "scene.h"

long ObjectsNum=0;
long NumGran=0;
long NumVertex=0;
int  NumLightSrc=0;
OBJECTPARAM   *objarray = 0;
GRANPARAM     *grandef = 0;
```

```
double diffuse;
double reflect;
double specular;
double p;              //показатель Фонга
double through;        //прозрачность
double r,g,b;          //цвет
double radius;         //радиус оболочки объекта
double xo,yo,zo;       //центр оболочки
};
struct GRANPARAM       //параметры грани
{
long start;            //индекс первой вершины грани
long nv;               //число вершин
double radius;         //радиус оболочки грани
double xo,yo,zo;       //центр оболочки
};
struct LIGHTSRCPARAM //параметры точечного источника света
{
VERTEX pos;            //расположение
double r,g,b;          //цвет
};
extern long ObjectsNum;     //количество объектов
extern long NumGran;        //граней
extern long NumVertex;      //вершин
extern int  NumLightSrc;    //источников света
extern OBJECTPARAM *objarray;     //массив объектов
extern GRANPARAM *grandef;        //массив граней
extern VERTEX *vg;                //массив вершин
extern LIGHTSRCPARAM *lightsrc;   //источники света
extern VERTEX camerapos,          //положение камеры
              cameradir;          //направление обзора
extern double zProjectionPlane;   //положение плоскости проецирования
                                  //на оси Z видовых координат
extern double SpaceColorR,        //цвет свободного пространства
              SpaceColorG,
              SpaceColorB;
extern int  OpenArrays_SCE(long no,long ng,long nv,int nl);
extern void CloseArrays_SCE(void);
extern void SetColor_SCE(double r,double g,double b);
extern void SetMaterial_SCE(double ambient,
                            double diffuse,
                            double reflect,
                            double specular,
                            double p,
                            double through);
```

```
extern void SetCameraPosition_SCE(double x,double y,double z,
                                  double dirx,double diry,
                                  double dirz,double zprojplane);
extern int AddLightSource_SCE(double r,double g,double b,
                              double x,double y,double z);
extern int AddTriangleGran_SCE(double x1,double y1,double z1,
                               double x2,double y2,double z2,
                               double x3,double y3,double z3,
                               int obj);
extern int AddQuadGran_SCE(double x1,double y1,double z1,
                           double x2,double y2,double z2,
                           double x3,double y3,double z3,
                           double x4,double y4,double z4,
                           int obj);
extern int AddPyramid_SCE(double x0,double y0,double z0,
                          double x1,double y1,double z1,
                          double x2,double y2,double z2,
                          double x3,double y3,double z3,
                          double x4,double y4,double z4);
extern int AddPrisma4_SCE(double x1,double y1,
                          double x2,double y2,
                          double x3,double y3,
                          double x4,double y4,
                          double zottom,double ztop);
extern int AddSphere_SCE(double x,double y,double z,
                         double r,int db,int dl);
extern int AddCylinder_SCE(double x,double y,double z,
                           double r,double h,int dl);
extern int CopyObjectAndShift_SCE(double sx,double sy,double sz);
#endif
```

Файл scene.cpp :

```
//--------(c) Copyright Порев В.Н.----------
#include <math.h>
#include "scene.h"

long ObjectsNum=0;
long NumGran=0;
long NumVertex=0;
int  NumLightSrc=0;
OBJECTPARAM   *objarray = 0;
GRANPARAM     *grandef = 0;
```

```
double diffuse;
double reflect;
double specular;
double p;              //показатель Фонга
double through;        //прозрачность
double r,g,b;          //цвет
double radius;         //радиус оболочки объекта
double xo,yo,zo;       //центр оболочки
};
struct GRANPARAM       //параметры грани
{
long start;            //индекс первой вершины грани
long nv;               //число вершин
double radius;         //радиус оболочки грани
double xo,yo,zo;       //центр оболочки
};
struct LIGHTSRCPARAM //параметры точечного источника света
{
VERTEX pos;            //расположение
double r,g,b;          //цвет
};
extern long ObjectsNum;     //количество объектов
extern long NumGran;        //граней
extern long NumVertex;      //вершин
extern int  NumLightSrc;    //источников света
extern OBJECTPARAM *objarray;     //массив объектов
extern GRANPARAM *grandef;        //массив граней
extern VERTEX *vg;                //массив вершин
extern LIGHTSRCPARAM *lightsrc;  //источники света
extern VERTEX camerapos,          //положение камеры
              cameradir;          //направление обзора
extern double zProjectionPlane;  //положение плоскости проецирования
                                  //на оси Z видовых координат
extern double SpaceColorR,        //цвет свободного пространства
              SpaceColorG,
              SpaceColorB;
extern int  OpenArrays_SCE(long no,long ng,long nv,int nl);
extern void CloseArrays_SCE(void);
extern void SetColor_SCE(double r,double g,double b);
extern void SetMaterial_SCE(double ambient,
                            double diffuse,
                            double reflect,
                            double specular,
                            double p,
                            double through);
```

```
extern void SetCameraPosition_SCE(double x,double y,double z,
                                  double dirx,double diry,
                                  double dirz,double zprojplane);
extern int AddLightSource_SCE(double r,double g,double b,
                              double x,double y,double z);
extern int AddTriangleGran_SCE(double x1,double y1,double z1,
                               double x2,double y2,double z2,
                               double x3,double y3,double z3,
                               int obj);
extern int AddQuadGran_SCE(double x1,double y1,double z1,
                           double x2,double y2,double z2,
                           double x3,double y3,double z3,
                           double x4,double y4,double z4,
                           int obj);
extern int AddPyramid_SCE(double x0,double y0,double z0,
                          double x1,double y1,double z1,
                          double x2,double y2,double z2,
                          double x3,double y3,double z3,
                          double x4,double y4,double z4);
extern int AddPrisma4_SCE(double x1,double y1,
                          double x2,double y2,
                          double x3,double y3,
                          double x4,double y4,
                          double zottom,double ztop);
extern int AddSphere_SCE(double x,double y,double z,
                         double r,int db,int dl);
extern int AddCylinder_SCE(double x,double y,double z,
                           double r,double h,int dl);
extern int CopyObjectAndShift_SCE(double sx,double sy,double sz);
#endif
```

Файл `scene.cpp` :

```
//--------(c) Copyright Порев В.Н.----------
#include <math.h>
#include "scene.h"

long ObjectsNum=0;
long NumGran=0;
long NumVertex=0;
int  NumLightSrc=0;
OBJECTPARAM   *objarray = 0;
GRANPARAM     *grandef = 0;
```

```
VERTEX        *vg = 0;
LIGHTSRCPARAM *lightsrc = 0;
VERTEX camerapos,
       cameradir;
double zProjectionPlane = -200;
double SpaceColorR = 0.9,
       SpaceColorG = 0.9,
       SpaceColorB = 0.9;
static long MaximumObjectsNum = 0,
            MaximumGranNum = 0,
            MaximumVertexNum = 0,
            MaximumLightSrc = 0;
static OBJECTPARAM currentparam;    //текущие свойства объекта
static long ObjectCopyNum=-1;       //номер копируемого объекта

static void EmptyScene_SCE(void);
static void Obolochka_SCE(double *r,double *x,double *y,double *z,
                          long start,long nv);
static void SpherePoint_SCE(VERTEX *v, double R, int b, int l);
static void CylinderPoint_SCE(VERTEX *v,double R,double H,
                              int s,int l);

//---создание рабочих массивов описания сцены------
int OpenArrays_SCE(long no,long ng,long nv,int nl)
{
CloseArrays_SCE();
EmptyScene_SCE();
objarray = new OBJECTPARAM[no];
grandef  = new GRANPARAM[ng];
vg       = new VERTEX[nv];
lightsrc = new LIGHTSRCPARAM[nv];
if ((objarray==0)||(grandef==0)||(vg==0)||(lightsrc==0))
   {
   CloseArrays_SCE();
   return 0;           //ошибка
   }
MaximumObjectsNum = no;
MaximumGranNum = ng;
MaximumVertexNum = nv;
MaximumLightSrc = nl;
return 1;
}
```

```
void CloseArrays_SCE(void)
{
if (lightsrc) delete []lightsrc;
if (vg)       delete []vg;
if (grandef)  delete []grandef;
if (objarray) delete []objarray;
objarray = 0;
grandef =  0;
vg =       0;
lightsrc = 0;
}

void EmptyScene_SCE(void)
{
ObjectsNum=NumGran=NumVertex=NumLightSrc=0;
ObjectCopyNum=-1;
//---------задаем материал объекта по умолчанию-------
currentparam.start=0;
currentparam.ng=0;
currentparam.ambient=0;
currentparam.diffuse=1;
currentparam.reflect=0;
currentparam.specular=0;
currentparam.p=20;
currentparam.through=0;
currentparam.r=1;
currentparam.g=1;
currentparam.b=1;
currentparam.radius=-1;
}

void SetColor_SCE(double r,double g,double b)
{
currentparam.r=r;
currentparam.g=g;
currentparam.b=b;
}

void SetMaterial_SCE(double ambient,
                     double diffuse,
                     double reflect,
                     double specular,
                     double p,
                     double through)
```

```
{
currentparam.ambient  = ambient;
currentparam.diffuse  = diffuse;
currentparam.reflect  = reflect;
currentparam.specular = specular;
currentparam.p = p;
currentparam.through  = through;
}

void SetCameraPosition_SCE(double x,double y,double z,
                           double dirx,double diry,double dirz,
                           double zprojplane)
{
camerapos.x = x;
camerapos.y = y;
camerapos.z = z;
cameradir.x = dirx;
cameradir.y = diry;
cameradir.z = dirz;
zProjectionPlane = -zprojplane;
}

int AddLightSource_SCE(double r,double g,double b,
                       double x,double y,double z)
{
if (NumLightSrc+1 >= MaximumLightSrc) return 0;
lightsrc[NumLightSrc].pos.x = x;
lightsrc[NumLightSrc].pos.y = y;
lightsrc[NumLightSrc].pos.z = z;
lightsrc[NumLightSrc].r = r;
lightsrc[NumLightSrc].g = g;
lightsrc[NumLightSrc].b = b;
NumLightSrc++;
return 1;
}
//------добавляем в сцену треугольную грань------------
//----obj=1, если грань как самостоятельный объект-------
int AddTriangleGran_SCE(double x1,double y1,double z1,
                        double x2,double y2,double z2,
                        double x3,double y3,double z3,
                        int obj)
```

```
{
if (ObjectsNum+1 >= MaximumObjectsNum) return 0;
if (NumGran+1 >= MaximumGranNum) return 0;
if (NumVertex+3 >= MaximumVertexNum) return 0;

if (obj)
   {
   currentparam.start = NumGran;
   currentparam.ng = 1;
   }
else currentparam.ng++;
currentparam.radius = -1;      //оболочка не определена
objarray[ObjectsNum] = currentparam;
grandef[NumGran].start = NumVertex;
grandef[NumGran].nv = 3;
vg[NumVertex].x = x1;
vg[NumVertex].y = y1;
vg[NumVertex].z = z1;
vg[NumVertex+1].x = x2;
vg[NumVertex+1].y = y2;
vg[NumVertex+1].z = z2;
vg[NumVertex+2].x = x3;
vg[NumVertex+2].y = y3;
vg[NumVertex+2].z = z3;
Obolochka_SCE(&grandef[NumGran].radius,
              &grandef[NumGran].xo,
              &grandef[NumGran].yo,
              &grandef[NumGran].zo,
              grandef[NumGran].start,3);
if (obj)
   {
   Obolochka_SCE(&objarray[ObjectsNum].radius,
                 &objarray[ObjectsNum].xo,
                 &objarray[ObjectsNum].yo,
                 &objarray[ObjectsNum].zo,
                 grandef[NumGran].start,3);
   ObjectCopyNum=ObjectsNum; //возможно, объект будет копироваться
   ObjectsNum++;
   }
NumGran++;
NumVertex += 3;
return 1;
}
```

```
int AddQuadGran_SCE(double x1,double y1,double z1,
                    double x2,double y2,double z2,
                    double x3,double y3,double z3,
                    double x4,double y4,double z4,
                    int obj)
{
if (ObjectsNum+1 >= MaximumObjectsNum) return 0;
if (NumGran+1 >= MaximumGranNum) return 0;
if (NumVertex+4 >= MaximumVertexNum) return 0;

if (obj)
   {
   currentparam.start = NumGran;
   currentparam.ng = 1;
   }
else currentparam.ng++;
currentparam.radius = -1;
objarray[ObjectsNum] = currentparam;
grandef[NumGran].start = NumVertex;
grandef[NumGran].nv = 4;
vg[NumVertex].x = x1;
vg[NumVertex].y = y1;
vg[NumVertex].z = z1;
vg[NumVertex+1].x = x2;
vg[NumVertex+1].y = y2;
vg[NumVertex+1].z = z2;
vg[NumVertex+2].x = x3;
vg[NumVertex+2].y = y3;
vg[NumVertex+2].z = z3;
vg[NumVertex+3].x = x4;
vg[NumVertex+3].y = y4;
vg[NumVertex+3].z = z4;
Obolochka_SCE(&grandef[NumGran].radius,
              &grandef[NumGran].xo,
              &grandef[NumGran].yo,
              &grandef[NumGran].zo,
              grandef[NumGran].start,4);
if (obj)
   {
   Obolochka_SCE(&objarray[ObjectsNum].radius,
                 &objarray[ObjectsNum].xo,
                 &objarray[ObjectsNum].yo,
                 &objarray[ObjectsNum].zo,
                 grandef[NumGran].start,4);
```

```
   ObjectCopyNum=ObjectsNum;
   ObjectsNum++;
   }
NumGran++;
NumVertex += 4;
return 1;
}

//----------x0,y0,z0 - это координаты вершины пирамиды-----
//----------x1,y1,z1 - x4,y4,z4 - основание-----------------
int AddPyramid_SCE(double x0,double y0,double z0,
                   double x1,double y1,double z1,
                   double x2,double y2,double z2,
                   double x3,double y3,double z3,
                   double x4,double y4,double z4)
{
long st;

st = NumVertex;
currentparam.start = NumGran;
currentparam.ng = 0;
AddQuadGran_SCE(x1,y1,z1,
                x2,y2,z2,
                x3,y3,z3,
                x4,y4,z4,0);
AddTriangleGran_SCE(x0,y0,z0,
                    x1,y1,z1,
                    x2,y2,z2,0);
AddTriangleGran_SCE(x0,y0,z0,
                    x2,y2,z2,
                    x3,y3,z3,0);
AddTriangleGran_SCE(x0,y0,z0,
                    x3,y3,z3,
                    x4,y4,z4,0);
AddTriangleGran_SCE(x0,y0,z0,
                    x4,y4,z4,
                    x1,y1,z1,0);
Obolochka_SCE(&objarray[ObjectsNum].radius,
              &objarray[ObjectsNum].xo,
              &objarray[ObjectsNum].yo,
              &objarray[ObjectsNum].zo,
              grandef[objarray[ObjectsNum].start].start,
              NumVertex-st);
```

```
ObjectCopyNum=ObjectsNum;
ObjectsNum++;
return 1;
}

int AddPrisma4_SCE(double x1,double y1,
                   double x2,double y2,
                   double x3,double y3,
                   double x4,double y4,
                   double zottom,double ztop)
{
long st;

st = NumVertex;
currentparam.start = NumGran;
currentparam.ng = 0;
AddQuadGran_SCE(x1,y1,zottom,
                x2,y2,zottom,
                x2,y2,ztop,
                x1,y1,ztop,0);
AddQuadGran_SCE(x2,y2,zottom,
                x3,y3,zottom,
                x3,y3,ztop,
                x2,y2,ztop,0);
AddQuadGran_SCE(x3,y3,zottom,
                x4,y4,zottom,
                x4,y4,ztop,
                x3,y3,ztop,0);
AddQuadGran_SCE(x4,y4,zottom,
                x1,y1,zottom,
                x1,y1,ztop,
                x4,y4,ztop,0);
AddQuadGran_SCE(x1,y1,zottom,
                x2,y2,zottom,
                x3,y3,zottom,
                x4,y4,zottom,0);
AddQuadGran_SCE(x1,y1,ztop,
                x2,y2,ztop,
                x3,y3,ztop,
                x4,y4,ztop,0);
Obolochka_SCE(&objarray[ObjectsNum].radius,
              &objarray[ObjectsNum].xo,
              &objarray[ObjectsNum].yo,
```

```
                grandef[objarray[ObjectsNum].start].start,
                NumVertex-st);
ObjectCopyNum=ObjectsNum;
ObjectsNum++;
return 1;
}

int AddCylinder_SCE(double x,double y,double z,
                    double r,double h,int dl)
{
int l;
long st;
VERTEX v[4];

st = NumVertex;
currentparam.start = NumGran;
currentparam.ng = 0;
for (l=0;l<360;l+=dl)
    {
    CylinderPoint_SCE(&v[0], r, h, 0, l);
    CylinderPoint_SCE(&v[1], r, h, 100, l);
    CylinderPoint_SCE(&v[2], r, h, 100, l+dl);
    CylinderPoint_SCE(&v[3], r, h, 0, l+dl);
    AddQuadGran_SCE(v[0].x+x, v[0].y+y, v[0].z+z,
                    v[1].x+x, v[1].y+y, v[1].z+z,
                    v[2].x+x, v[2].y+y, v[2].z+z,
                    v[3].x+x, v[3].y+y, v[3].z+z,
                    0);
    }
Obolochka_SCE(&objarray[ObjectsNum].radius,
                &objarray[ObjectsNum].xo,
                &objarray[ObjectsNum].yo,
                &objarray[ObjectsNum].zo,
                grandef[objarray[ObjectsNum].start].start,
                NumVertex-st);
ObjectCopyNum=ObjectsNum;
ObjectsNum++;
return 1;
}

//-----копирование ранее введенного объекта и сдвиг-----
int CopyObjectAndShift_SCE(double sx,double sy,double sz)
{
long startgran,numgran,startvertex,numvertex,i,ii;
```

```
if ((ObjectCopyNum < 0)||(ObjectCopyNum>=ObjectsNum)) return 0;
objarray[ObjectsNum] = objarray[ObjectCopyNum];
objarray[ObjectsNum].start = NumGran;  //первая свободная грань
objarray[ObjectsNum].xo += sx;       //сдвигаем оболочку объекта
objarray[ObjectsNum].yo += sy;
objarray[ObjectsNum].zo += sz;
startgran = objarray[ObjectCopyNum].start; //первая копируемая грань
numgran = objarray[ObjectCopyNum].ng;      //число граней
for (i=0;i<numgran;i++)
    {
    grandef[NumGran] = grandef[startgran+i];
    grandef[NumGran].start = NumVertex; //первая свободная вершина
    grandef[NumGran].xo += sx;          //сдвигаем оболочку грани
    grandef[NumGran].yo += sy;
    grandef[NumGran].zo += sz;
    NumGran++;
    startvertex = grandef[startgran+i].start;
    numvertex = grandef[startgran+i].nv;
    for (ii=0; ii<numvertex; ii++)        //сдвигаем вершины
        {
        vg[NumVertex] = vg[startvertex + ii];
        vg[NumVertex].x += sx;
        vg[NumVertex].y += sy;
        vg[NumVertex].z += sz;
        NumVertex++;
        }
    }
ObjectsNum++;
return 1;
}

//-----------координаты точки поверхности шара-----------------
void SpherePoint_SCE(VERTEX *v, double R, int b, int l)
{
double B,L;

B = (double)b*M_PI/180.0;
L = (double)l*M_PI/180.0;
v->x = R*cos(B)*sin(L);
v->y = R*cos(B)*cos(L);
v->z = R*sin(B);
}
```

```
void CylinderPoint_SCE(VERTEX *v,double R,double H,int s,int l)
{
double L;

L = (double)l*M_PI/180.0;
v->x = R*sin(L);
v->y = R*cos(L);
v->z = (double)s*H/100.0;
}

//--------вычисление радиуса и центра оболочки------
//-----start - начальная вершина объекта (грани)------
void Obolochka_SCE(double *r,double *x,double *y,double *z,
                   long start,long nv)
{
long i;
double xo,yo,zo,rad,r1;
double minx,maxx,miny,maxy,minz,maxz;

minx=maxx=vg[start].x;
miny=maxy=vg[start].y;
minz=maxz=vg[start].z;
for (i=1;i<nv;i++)
    {
    if (minx > vg[start+i].x) minx = vg[start+i].x;
    if (maxx < vg[start+i].x) maxx = vg[start+i].x;
    if (miny > vg[start+i].y) miny = vg[start+i].y;
    if (maxy < vg[start+i].y) maxy = vg[start+i].y;
    if (minz > vg[start+i].z) minz = vg[start+i].z;
    if (maxz < vg[start+i].z) maxz = vg[start+i].z;
    }
//---------центр оболочки=центр параллелепипеда----
xo = 0.5*(maxx+minx);
yo = 0.5*(maxy+miny);
zo = 0.5*(maxz+minz);
//--------начальное приближение радиуса оболочки---
rad = 0.5*(maxx-minx);
r1 = 0.5*(maxy-miny);
if (rad < r1) rad=r1;
r1 = 0.5*(maxz-minz);
if (rad < r1) rad=r1;
//--------уточняем радиус----------
for (i=0;i<nv;i++)
```

```
    {
    r1 = sqrt((xo-vg[start+i].x)*(xo-vg[start+i].x) +
              (yo-vg[start+i].y)*(yo-vg[start+i].y) +
              (zo-vg[start+i].z)*(zo-vg[start+i].z));
    if (rad < r1) rad = r1;
    }
*x = xo;
*y = yo;
*z = zo;
*r = rad;
}
```

В модуле RAYTRACE реализована трассировка лучей методом локальных координат.

Файл raytrace.h:

```
extern void Rendering_RAYTR(HDC hdc, int cx, int cy);
```

Файл raytrace.cpp:

```
//--------(c) Copyright Порев В.Н.-----------
#define STRICT
#include <windows.h>
#include <math.h>
#include <mem.h>
#include "scene.h"
#include "raytrace.h"

static VERTEX pgn[4];     //массив вершин текущей грани

//--------служебные функции трассировки--------------
static void SetKoordTransform_RAYTR(double *M,VERTEX *rv,
                                    double x,double y,double z);
static void CreateStartRay_RAYTR(double *M,double *MV,int x,int y);
static void Ray_RAYTR(double *r,double *g,double *b,
                      double *M,long nobj,long ngran,int level);
static void DiffuseRay_RAYTR(double *r,double *g,double *b,
                             double *M,long nobj,long ngran);
static void SpecularRay_RAYTR(double *r,double *g,double *b,
                              double *M,int ng,double p);
static BOOL LightSourceIsVisible_RAYTR(double *r,double *g,
                                       double *b,double *M,
                                       double zl,int srcnum,
                                       int ngran);
```

```
static BOOL NearestGranPoint_RAYTR(double *M,double *znear,
                                   long *no,long *ng,long ngr);
static BOOL InObolochkaGran_RAYTR(double *M,int ng);
static BOOL InObolochkaObject_RAYTR(double *M,int no);
static BOOL IntersectPolygon_RAYTR(double *zp,int nv);
static void MatrixAxB_RAYTR(double *dest,double *A, double *B);
static void MatrixShift_RAYTR(double *dest,double *src, double dz);
static void TransformKoord_RAYTR(double *M,VERTEX *dest,
                                 VERTEX *src);
static void NormalToGran_RAYTR(double *M,VERTEX *normal,int ng);
static void NormalVector_RAYTR(VERTEX *nv,
                               VERTEX v1, VERTEX v2, VERTEX v3);
static void Normalize_RAYTR(VERTEX *v);
static void ReflectionVector_RAYTR(VERTEX *vr,VERTEX *n);

//-------главный цикл создания изображения-------------
void Rendering_RAYTR(HDC hdc, int cx, int cy)
{
int x,y;
double M[16],MV[16];              //матрицы преобразований
double r=0,g=0,b=0;

SetKoordTransform_RAYTR(MV,&cameradir,      //видовая матрица
                        camerapos.x,    //будет учитывать
                        camerapos.y,    //положение камеры
                        camerapos.z);
for (y=-cy/2; y<cy/2; y++)
  for (x=-cx/2; x<cx/2; x++)
    {
    CreateStartRay_RAYTR(M,MV,x,y);
    Ray_RAYTR(&r,&g,&b,  M, -1, -1, 0);     //первичный луч
    r *= 255.0;
    g *= 255.0;
    b *= 255.0;
    SetPixel(hdc,x+cx/2, y+cy/2,
             RGB((BYTE)r,(BYTE)g,(BYTE)b));
    }
}

//---формируем матрицу преобразования для локальной системы---
//--центр (0,0,0) локальной системы координат находится---
//------------в начальной точке луча (x,y,z),-------------
//-------а ось Z направлена противоположно лучу-----------
//-----вектор (-rv) указывает новое направление оси Z-----
```

```
void SetKoordTransform_RAYTR(double *M,VERTEX *rv,
                             double x,double y,double z)
{
double R,r;
double csA,snA,csB,snB;

//-------сначала коэффициенты единичной матрицы---------
for (int i=0;i<16;i++) M[i]=0;
M[0]=M[5]=M[10]=M[15]=1;
//-----а теперь формируем матрицу преобразования--------
if ((rv->x == 0)&&(rv->y == 0))    //нужен только сдвиг
   {
   if (rv->z <= 0)
      {
      M[11] = -z;
      return;
      }
   else
      {
      M[5] = M[10] = -1;
      M[11] = z;
      return;
      }
   }
//--------формируем матрицу поворота-------------------
R = sqrt(rv->x*rv->x + rv->y*rv->y + rv->z*rv->z);
r = sqrt(rv->x*rv->x + rv->y*rv->y);
csB = -rv->z/R;
snB = r/R;
csA = -rv->y/r;
snA = -rv->x/r;
M[0] = csA;
M[1] = -snA;
M[2] = 0;
M[3] = -M[0]*x - M[1]*y;
M[4] = snA*csB;
M[5] = csA*csB;
M[6] = -snB;
M[7] = -M[4]*x - M[5]*y - M[6]*z;
M[8] = snA*snB;
M[9] = csA*snB;
M[10] = csB;
M[11] = -M[8]*x - M[9]*y - M[10]*z;
}
```

```
//---формируем матрицу для первичного луча - из камеры---
void CreateStartRay_RAYTR(double *M,double *MV,int x,int y)
{
VERTEX dir;

dir.x = x;
dir.y = y;
dir.z = zProjectionPlane;     //точка в плоскости проецирования
SetKoordTransform_RAYTR(M, &dir, 0,0,0);  //из точки схода лучей
MatrixAxB_RAYTR(M, M, MV);    //учитываем видовое преобразование
}

//------------основная процедура отслеживания лучей------------
void Ray_RAYTR(double *r,double *g,double *b,
               double *M,long nobj,long ngran,int level)

{
long no,ng;
double z;
VERTEX normal,dir;
double Md[16],Mr[16],Ms[16],Mt[16];
double rres,gres,bres,
       robj,gobj,bobj,
       rdif=0,gdif=0,bdif=0,
       rref=0,gref=0,bref=0,
       rspe=0,gspe=0,bspe=0,
       rthr=0,gthr=0,bthr=0;

if (level > 20)     //защита от зацикливания
   {
   *r = *g = *b = 0.5;
   return;
   }
if (NearestGranPoint_RAYTR(M,&z,&no,&ng,ngran)) //луч пересекает
                                                //объект
   {
   robj = objarray[no].r;             //заданный цвет объекта
   gobj = objarray[no].g;
   bobj = objarray[no].b;
   NormalToGran_RAYTR(M,&normal,ng);     //вектор нормали
   if (objarray[no].diffuse > 0)
      {
      SetKoordTransform_RAYTR(Md,&normal,0,0,z);
```

```
    MatrixAxB_RAYTR(Md, Md, M);
    DiffuseRay_RAYTR(&rdif,&gdif,&bdif, Md,no,ng);
    }
if (objarray[no].reflect > 0)
    {
    ReflectionVector_RAYTR(&dir, &normal);
    SetKoordTransform_RAYTR(Mr,&dir,0,0,z);
    MatrixAxB_RAYTR(Mr, Mr, M);
    Ray_RAYTR(&rref,&gref,&bref, Mr, no, ng, level+1);
    }
if (objarray[no].specular > 0)
    {
    ReflectionVector_RAYTR(&dir, &normal);
    SetKoordTransform_RAYTR(Ms,&dir,0,0,z);
    MatrixAxB_RAYTR(Ms, Ms, M);
    SpecularRay_RAYTR(&rspe,&gspe,&bspe, Ms,ng, objarray[no].p);
    }
if (objarray[no].through > 0)
    {
    MatrixShift_RAYTR(Mt, M, z);  //продолжаем вдоль оси Z
    Ray_RAYTR(&rthr,&gthr,&bthr, Mt, no, ng, level+1);
    }
rres = robj*objarray[no].ambient  +
       rdif*objarray[no].diffuse  +
       rref*objarray[no].reflect  +
       rspe*objarray[no].specular +
       rthr*robj*objarray[no].through;
gres = gobj*objarray[no].ambient  +
       gdif*objarray[no].diffuse  +
       gref*objarray[no].reflect  +
       gspe*objarray[no].specular +
       gthr*gobj*objarray[no].through;
bres = bobj*objarray[no].ambient  +
       bdif*objarray[no].diffuse  +
       bref*objarray[no].reflect  +
       bspe*objarray[no].specular +
       bthr*bobj*objarray[no].through;
if (rres > 1) rres=1;
if (gres > 1) gres=1;
if (bres > 1) bres=1;
*r = rres;
*g = gres;
*b = bres;
}
```

```
else       //луч ушел в свободное пространство
   {
   *r = SpaceColorR;
   *g = SpaceColorG;
   *b = SpaceColorB;
   }
}

//-------моделирование диффузного отражения-------
//-------лучи направляем на источники света--------
void DiffuseRay_RAYTR(double *r,double *g,double *b,
                      double *M,long nobj,long ngran)
{
VERTEX v,vl;
int i;
double ro,go,bo,
       rl,gl,bl,
       rres,gres,bres;
double Kd;
double Ml[16];

ro = objarray[nobj].r;              //исходный цвет объекта
go = objarray[nobj].g;
bo = objarray[nobj].b;
rres=gres=bres=0;
for (i=0; i<NumLightSrc; i++)
    {
    TransformKoord_RAYTR(M,&v, &lightsrc[i].pos);
    Normalize_RAYTR(&v);            //вектор на источник света
    SetKoordTransform_RAYTR(Ml,&v,0,0,0);
    MatrixAxB_RAYTR(Ml, Ml, M);
    rl = lightsrc[i].r;
    gl = lightsrc[i].g;
    bl = lightsrc[i].b;
    TransformKoord_RAYTR(Ml,&vl, &lightsrc[i].pos);
    if (LightSourceIsVisible_RAYTR(&rl,&gl,&bl, Ml,vl.z, i,ngran))
       {
       Kd = -v.z;
       if (Kd < 0) Kd=0;
       if (Kd > 1) Kd=1;
       rres += Kd*ro*rl;
       gres += Kd*go*gl;
       bres += Kd*bo*bl;
       }
    }
```

```
if (rres > 1) rres=1;
if (gres > 1) gres=1;
if (bres > 1) bres=1;
*r = rres;
*g = gres;
*b = bres;
}

//-----------моделирование зеркальных бликов-----------------
//-----p - показатель степени зеркальности Фонга (1..200)----
//--------эта функция почти идентична предудущей, но---------
//--------для улучшения восприятия сделана отдельной---------
void SpecularRay_RAYTR(double *r,double *g,double *b,
                       double *M,int ng,double p)
{
VERTEX v,vl;
int i;
double rl,gl,bl,rres,gres,bres;
double Kd;
double Ml[16];

rres=gres=bres=0;
for (i=0; i<NumLightSrc; i++)
    {
    TransformKoord_RAYTR(M,&v, &lightsrc[i].pos);
    Normalize_RAYTR(&v);        //вектор на источник света
    SetKoordTransform_RAYTR(Ml,&v,0,0,0);
    MatrixAxB_RAYTR(Ml, Ml, M);
    rl = lightsrc[i].r;
    gl = lightsrc[i].g;
    bl = lightsrc[i].b;
    TransformKoord_RAYTR(M,&vl, &lightsrc[i].pos);
    if (LightSourceIsVisible_RAYTR(&rl,&gl,&bl, Ml,vl.z, i,ng))
       {
       Kd = -v.z;           //cos
       if (Kd < 0) Kd=0;
       if (Kd > 1) Kd=1;
       Kd = pow(Kd,p);      //cos в степени p
       rres += Kd*rl;
       gres += Kd*gl;
       bres += Kd*bl;
       }
    }
```

```
if (rres > 1) rres=1;
if (gres > 1) gres=1;
if (bres > 1) bres=1;
*r = rres;
*g = gres;
*b = bres;
}

//------виден ли источник света из данной точки?------
//---------направляем луч на источник света-----------
//--------zl - расстояние до источника света----------
BOOL LightSourceIsVisible_RAYTR(double *r,double *g,double *b,
                                double *M,double zl,
                                int srcnum,int ngran)
{
long no,ng;
double z;
double rres,gres,bres;
double Mt[16];

rres = *r;
gres = *g;
bres = *b;
if (NearestGranPoint_RAYTR(M,&z,&no,&ng,ngran)) //есть пересечение
   if (z > zl)     //источник света ближе
      {
      if (objarray[no].through < 0.01)     //на пути луча
         {                                 //непрозрачная грань
         *r = 0;
         *g = 0;
         *b = 0;
         return FALSE;
         }
      else            //сквозь прозрачную грань нужно идти дальше
         {
         rres *= objarray[no].r*objarray[no].through;
         gres *= objarray[no].g*objarray[no].through;
         bres *= objarray[no].b*objarray[no].through;
         MatrixShift_RAYTR(Mt, M, z);   //продолжаем вдоль луча
         LightSourceIsVisible_RAYTR(&rres,&gres,&bres,
                                    Mt, zl-z, srcnum, ng);
         }
   }
```

```
*r = rres;
*g = gres;
*b = bres;
return TRUE;       //нет препятствий на пути луча света
}

//---координата Z точки пересечения луча с ближайшей гранью----
//возвращаются также номер пересекаемого объекта (no) и грани (ng)
//--------ngr указывает грань, из которой луч выпущен---------
BOOL NearestGranPoint_RAYTR(double *M,double *znear,
                            long *no,long *ng,long ngr)
{
long i,ii,iii,numgr;
long objnumber=-1,grannumber=-1;
double z,zres;
BOOL first=TRUE;

for (i=0; i<ObjectsNum; i++)              //цикл объектов
    {
    if (!InObolochkaObject_RAYTR(M,i)) continue;
    for (ii=0; ii<objarray[i].ng; ii++)  //цикл граней
        {
        numgr = objarray[i].start + ii;  //номер текущей грани
        if (numgr == ngr) continue;      //из этой грани исходит луч
        if (!InObolochkaGran_RAYTR(M,numgr)) continue;
        for (iii=0; iii<grandef[numgr].nv; iii++)
            TransformKoord_RAYTR(M,
                                 &pgn[iii],
                                 &vg[grandef[numgr].start + iii]);
        if (!IntersectPolygon_RAYTR(&z, grandef[numgr].nv))
           continue;
        if (z >= 0) continue;
        if (first)
           {
           objnumber=i;
           grannumber=numgr;
           zres = z; first=FALSE;
           }
        else
           {
           if (zres < z)
              {
              zres = z;
```

```
                objnumber=i;
                grannumber=numgr;
                }
            }
        }
    }
if (grannumber >= 0)        //грань найдена
   {
   *znear = zres;       //координата точки пересечения
   *no = objnumber;     //номер пересекаемого объекта
   *ng = grannumber;    //номер ближайшей видимой грани
   return TRUE;
   }
return FALSE;
}

//--------пересекается ли лучом оболочка объекта?----------
BOOL InObolochkaObject_RAYTR(double *M,int no)
{
VERTEX v;
double r;

v.x = objarray[no].xo;
v.y = objarray[no].yo;
v.z = objarray[no].zo;
r = objarray[no].radius;
TransformKoord_RAYTR(M,&v,&v);
if (v.x*v.x + v.y*v.y <= r*r)
   return TRUE;
return FALSE;
}

//--------пересекается ли лучом оболочка грани?----------
BOOL InObolochkaGran_RAYTR(double *M,int ng)
{
VERTEX v;
double r;

v.x = grandef[ng].xo;
v.y = grandef[ng].yo;
v.z = grandef[ng].zo;
r = grandef[ng].radius;
TransformKoord_RAYTR(M,&v,&v);
if (v.x*v.x + v.y*v.y <= r*r)
```

```
   return TRUE;
return FALSE;
}

//определение координаты Z точки пересечения с полигоном
//-----вершины грани должны быть в массиве pgn[]--------
BOOL IntersectPolygon_RAYTR(double *zp,int nv)
{
int i,nhor,next,xmin,xmax,ymin,ymax;
double x[4],z[4];
double y1,y2;

xmin=xmax=pgn[0].x;
ymin=ymax=pgn[0].y;
for (i=1;i<nv;i++)
  {
  if (xmin > pgn[i].x) xmin = pgn[i].x;
  if (ymin > pgn[i].y) ymin = pgn[i].y;
  if (xmax < pgn[i].x) xmax = pgn[i].x;
  if (ymax < pgn[i].y) ymax = pgn[i].y;
  }
if ((xmin > 0)||(xmax < 0)||(ymin > 0)||(ymax < 0))
   return FALSE;
nhor = 0;
for (i=0; i<nv; i++)
  {
  next = i+1;
  if (next >= nv) next=0;
  y1 = pgn[i].y;
  y2 = pgn[next].y;
  if ((0 >= y1)&&(0 < y2)||(0 <= y1)&&(0 > y2))
     {
     x[nhor]=pgn[i].x-(pgn[next].x-pgn[i].x)*y1/(y2-y1);
     z[nhor]=pgn[i].z-(pgn[next].z-pgn[i].z)*y1/(y2-y1);
     nhor++;
     }
  }
if (nhor != 2) return FALSE;
if (x[1]==x[0])
   {
   if (x[1] == 0)
      {
      *zp = 0.5*(z[0]+z[1]);
      return TRUE;
      }
```

```
                objnumber=i;
                grannumber=numgr;
                }
             }
          }
       }
if (grannumber >= 0)        //грань найдена
   {
   *znear = zres;       //координата точки пересечения
   *no = objnumber;     //номер пересекаемого объекта
   *ng = grannumber;    //номер ближайшей видимой грани
   return TRUE;
   }
return FALSE;
}

//--------пересекается ли лучом оболочка объекта?----------
BOOL InObolochkaObject_RAYTR(double *M,int no)
{
VERTEX v;
double r;

v.x = objarray[no].xo;
v.y = objarray[no].yo;
v.z = objarray[no].zo;
r = objarray[no].radius;
TransformKoord_RAYTR(M,&v,&v);
if (v.x*v.x + v.y*v.y <= r*r)
   return TRUE;
return FALSE;
}

//--------пересекается ли лучом оболочка грани?----------
BOOL InObolochkaGran_RAYTR(double *M,int ng)
{
VERTEX v;
double r;

v.x = grandef[ng].xo;
v.y = grandef[ng].yo;
v.z = grandef[ng].zo;
r = grandef[ng].radius;
TransformKoord_RAYTR(M,&v,&v);
if (v.x*v.x + v.y*v.y <= r*r)
```

```
   return TRUE;
return FALSE;
}

//определение координаты Z точки пересечения с полигоном
//-----вершины грани должны быть в массиве pgn[]--------
BOOL IntersectPolygon_RAYTR(double *zp,int nv)
{
int i,nhor,next,xmin,xmax,ymin,ymax;
double x[4],z[4];
double y1,y2;

xmin=xmax=pgn[0].x;
ymin=ymax=pgn[0].y;
for (i=1;i<nv;i++)
  {
  if (xmin > pgn[i].x) xmin = pgn[i].x;
  if (ymin > pgn[i].y) ymin = pgn[i].y;
  if (xmax < pgn[i].x) xmax = pgn[i].x;
  if (ymax < pgn[i].y) ymax = pgn[i].y;
  }
if ((xmin > 0)||(xmax < 0)||(ymin > 0)||(ymax < 0))
   return FALSE;
nhor = 0;
for (i=0; i<nv; i++)
  {
  next = i+1;
  if (next >= nv) next=0;
  y1 = pgn[i].y;
  y2 = pgn[next].y;
  if ((0 >= y1)&&(0 < y2)||(0 <= y1)&&(0 > y2))
     {
     x[nhor]=pgn[i].x-(pgn[next].x-pgn[i].x)*y1/(y2-y1);
     z[nhor]=pgn[i].z-(pgn[next].z-pgn[i].z)*y1/(y2-y1);
     nhor++;
     }
  }
if (nhor != 2) return FALSE;
if (x[1]==x[0])
   {
   if (x[1] == 0)
      {
      *zp = 0.5*(z[0]+z[1]);
      return TRUE;
      }
```

```
   return FALSE;
   }
if (((x[0] <= 0)&&(x[1] >= 0))||
    ((x[0] >= 0)&&(x[1] <= 0)))
   {
   *zp = z[0]-x[0]*(z[1]-z[0])/(x[1]-x[0]);
   return TRUE;
   }
return FALSE;
}

//--------------перемножение матриц--------------------
void MatrixAxB_RAYTR(double *dest,double *A, double *B)
{
int indx,i,j,k;
double tmp[12];

for (indx=0; indx<12; indx++)
    {
    i = indx & 0xfc;
    j = indx & 3;
    tmp[indx]=0;
    for (k=0; k<4; k++)
        {
        tmp[indx] += A[i]*B[j];
        i++;
        j += 4;
        }
    }
memcpy(dest,tmp,12*sizeof(double));
dest[12]=dest[13]=dest[14]=0; dest[15]=1;
}

//--------------матрица сдвига по оси Z--------------------
void MatrixShift_RAYTR(double *dest,double *src, double dz)
{
memcpy(dest,src,16*sizeof(double));
dest[11] -= dz;
}

//-----трехмерное аффинное преобразование координат-------
void TransformKoord_RAYTR(double *M,VERTEX *dest,VERTEX *src)
{
VERTEX tmp;
```

```
tmp.x = M[0]*src->x + M[1]*src->y +  M[2]*src->z + M[3];
tmp.y = M[4]*src->x + M[5]*src->y +  M[6]*src->z + M[7];
tmp.z = M[8]*src->x + M[9]*src->y + M[10]*src->z + M[11];
*dest = tmp;
}

void NormalToGran_RAYTR(double *M,VERTEX *normal,int ng)
{
VERTEX v1,v2,v3;

TransformKoord_RAYTR(M,&v1, &vg[grandef[ng].start]);
TransformKoord_RAYTR(M,&v2, &vg[grandef[ng].start+1]);
TransformKoord_RAYTR(M,&v3, &vg[grandef[ng].start+2]);
NormalVector_RAYTR(normal,v1,v2,v3);
Normalize_RAYTR(normal);
}

//---------нормаль к треугольнику (v1, v2, v3)----------
void NormalVector_RAYTR(VERTEX *nv,VERTEX v1, VERTEX v2, VERTEX v3)
{
v2.x -= v1.x;
v2.y -= v1.y;
v2.z -= v1.z;
v3.x -= v1.x;
v3.y -= v1.y;
v3.z -= v1.z;
nv->z =  v2.x * v3.y - v3.x * v2.y;
nv->x =  v2.y * v3.z - v3.y * v2.z;
nv->y = -v2.x * v3.z + v3.x * v2.z;
if (nv->z < 0)    //выбираем нормаль видимой стороны грани
   {
   nv->x = -nv->x;
   nv->y = -nv->y;
   nv->z = -nv->z;
   }
}

void Normalize_RAYTR(VERTEX *v)
{
double R;

R = sqrt(v->x*v->x + v->y*v->y + v->z*v->z);
v->x /= R;
v->y /= R;
v->z /= R;
}
```

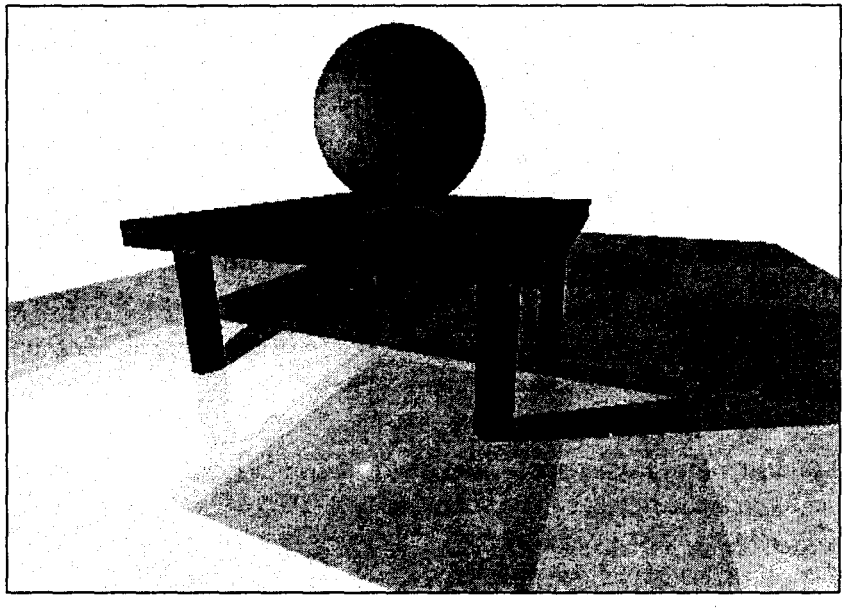
```
//----вектор луча зеркального отражения----------
//---падающий луч вдоль оси Z: (0, 0, 1)---------
void ReflectionVector_RAYTR(VERTEX *vr,VERTEX *n)
{
vr->x = n->x*n->z;
vr->y = n->y*n->z;
vr->z = n->z*n->z - 0.5;
}
```

Модули SCENE и RAYTRACE можно использовать в программах для Windows, например, следующим образом:

```
#include <windows.h>
#include "scene.h"
#include "raytrace.h"
. . . .
. . . .
HWND hWnd;
HDC  hdc;
RECT rc;

hdc = GetDC(hWnd);
GetClientRect(hWnd,&rc);      //определяем размеры области вывода
//-------создаем сцену------------
if (OpenArrays_SCE(100,3000,12000,10))
   {
   SetCameraPosition_SCE(300,600,270,-0.4,-1,-0.3,300); //камера
   AddLightSource_SCE(1,1,1,-150,400,300);       //источник света
//-----темно-коричневый стол--------
   SetColor_SCE(0.7,0.3,0);
   SetMaterial_SCE(0.5, 0.5, 0, 0, 20, 0);       //диффузный
   AddCylinder_SCE(-200,200,0,21,170,30);
   CopyObjectAndShift_SCE(400,0,0);
   CopyObjectAndShift_SCE(0,-400,0);
   CopyObjectAndShift_SCE(400,-400,0);
   SetMaterial_SCE(0.3, 0.5, 0.2, 0, 20, 0);     //полузеркальный
   AddPrisma4_SCE(-250,250, 250,250, 250,-250, -250,-250, 170,200);
//------голубой шар-------
   SetColor_SCE(0,1,1);
   SetMaterial_SCE(0.2, 0.8, 0, 0, 20, 0);       //диффузный
   AddSphere_SCE(0,0,320,120,6,6);
//-------слабо зеркальные клетки шахматного поля----
   SetColor_SCE(1,0.8,0.8);
   SetMaterial_SCE(0.5, 0.4, 0.1, 0, 20, 0);
```

```
   AddQuadGran_SCE(0,0,0, 400,0,0, 400,400,0, 0,400,0, 1);
   CopyObjectAndShift_SCE(-400,-400,0);
   CopyObjectAndShift_SCE(0,-800,0);
   CopyObjectAndShift_SCE(-400,-1200,0);
   CopyObjectAndShift_SCE(400,-400,0);
   CopyObjectAndShift_SCE(400,-1200,0);
   CopyObjectAndShift_SCE(-800,0,0);
   CopyObjectAndShift_SCE(-800,-800,0);
   SetColor_SCE(0.8,1,1);
   AddQuadGran_SCE(0,-400,0, 400,-400,0, 400,0,0, 0,0,0, 1);
   CopyObjectAndShift_SCE(-400,400,0);
   CopyObjectAndShift_SCE(0,-800,0);
   CopyObjectAndShift_SCE(-400,-400,0);
   CopyObjectAndShift_SCE(400,400,0);
   CopyObjectAndShift_SCE(400,-400,0);
   CopyObjectAndShift_SCE(-800,0,0);
   CopyObjectAndShift_SCE(-800,-800,0);
//--------отображение сцены--------
   Rendering_RAYTR(hdc, rc.right, rc.bottom);
//---------уничтожаем сцену--------
   CloseArrays_SCE();
   }
ReleaseDC(hWnd,hdc);
```

Результат работы программы приведен ниже.



Глоссарий

Альфа-канал — характеристика прозрачности.

Анимация (animation) — последовательность кадров, которые воспринимаются как кино.

Антиалиасинг (antialiasing) — устранение ступенчатого эффекта растровых изображений, сглаживание путем задания цветов отдельных угловых пикселов.

Аффинное преобразование — линейное преобразование, например, преобразование координат.

Битовый массив (bitmap) — растр, который сохраняется в памяти или на диске.

Векторизация (vectorization) — преобразование в векторную форму описания из растровой или другой формы.

Векторная графика — создание изображений на основе векторного описания отдельных объектов.

Окно (window) — фрагмент плоскости графического вывода. В операционной системе Windows это фундаментальное понятие, которое ассоциируется с выполняемой программой.

Видеоадаптер — устройство, с помощью которого непосредственно формируется изображение на экране монитора компьютерной системы. Создание изображения осуществляется на основе данных, которые присылаются из процессора и памяти.

Виртуальная реальность (virtual reality) — понятие, которое означает способность компьютерной системы создать для человека иллюзию действий в некотором пространстве.

Видовое преобразование (view transform) — преобразование координат согласно ракурсу показа пространственных объектов.

Воксел (voxel — volume picture element) — элемент объемного растра. Массивы *в.* используются для моделирования объемных трехмерных объектов.

Воксельный рендеринг — отображение на основе воксельной модели.

- ***В.р.*** прямого хода (forward rendering) — вывод точек в произвольном порядке с использованием Z-буфера.
- ***В.р.*** BTF (back-to-front) — сканирование объема начинается с самых дальних вокселов и заканчивается самыми близкими.
- ***В.р.*** FTB (front-to-back) — сканирование объема начинается с самых близких вокселов и заканчивается самыми дальними вокселами.

Гамма-коррекция — компенсация нелинейности функции яркости для экранов дисплеев телевизионного типа.

Геоинформационная система (ГИС) — компьютерная система, которая хранит в базе данных описание и обеспечивает анализ объектов, которые располагаются на поверхности Земли.

Графический интерфейс пользователя (Graphical User Interface, GUI) — набор графических элементов, которые предусмотрены для пользователей компьютерной системы для выполнения некоторых операций.

Гуро метод — способ закрашивания граней трехмерных объектов, который использует интерполяцию интенсивностей отражения света в вершинах граней.

Дизеринг (dithering) — иллюзия оттенка цвета, созданная смешением близко расположенных точек различных цветов.

Интерактивная компьютерная графика — понятие, которое использовалось для того, чтобы подчеркнуть наличие аппаратных и программных способов диалога с человеком в графической компьютерной системе.

Интерфейс графического устройства (Graphic Device Interface, GDI) — подсистема операционной системы Windows.

Компьютерная графика — создание изображений с помощью компьютера.

Контекст (context) — указывает место графического вывода. С контекстом ассоциируется поверхность отображения и структура данных, которые описывают основные параметры.

- ***К.*** графического устройства (device context) — фундаментальное понятие графики для функций API Windows.
- ***К.*** отображения (rendering context) — контекст для функций библиотеки OpenGL.

Линиатура — количество точек (линий) на единицу длины. Используется для характеристики растеризации методом дизеринга.

Метафайл (metafile) — описание изображения в файле, которое содержит операторы графики в соответствующей последовательности.

Морфинг (morphing) — методы преобразования формы объектов.

Муар — видеоэффект, узор, который возникает вследствие взаимодействия растровых структур изображения и растровых элементов устройства отображения.

Операционная система — программа, которая управляет ресурсами компьютера и другими программами. Пример *о.с.* — Windows, Unix, MacOS.

Палитра (pallete) — набор цветов, важных для определенного изображения.

Пиксел (pixel — picture element) — элемент растра.

Плоттер (plotter) — векторное устройство для отображения на бумаге.

Полигон (polygon) — многоугольник, фигура, которая ограничивается контуром связанных отрезков прямых.

Полилиния (polyline) — ломаная линия связанных отрезков прямых.

Принтер (printer) — устройство для печати, преимущественно растрового типа.

Растеризация (rasterization) — создание растрового изображения на основе векторного (или другого) описания элементов изображения.

Растровое изображение — изображение, созданное множеством близко расположенных точек различного цвета (пикселов).

Рендеринг (rendering) — процесс отображения информации в графическом виде. Как правило, это относится к созданию изображений трехмерных объектов.

Разрешающая способность растра (resolution) — характеристика растров и растровых устройств. Измеряется в количестве пикселов на единицу длины, например, в дюймах (dpi).

- Оптическая *р. сп. р.* — характеризует оптическую систему растровых устройств ввода-вывода.
- Интерполированная *р. сп. р.* — выше оптической благодаря интерполяции.

Система автоматизированного проектирования, САПР (Computer Aided Design, CAD) — система для проектирования сложных объектов с по-

мощью компьютера. Обычно функционирует в интерактивном режиме и широко использует компьютерную графику.

Сканер (scanner) — устройство для ввода графических изображений в компьютер.

Сплайн — кривая или поверхность специального типа, которая может использоваться для аппроксимации фрагментов линий или поверхностей сложной формы. Несколько связанных сплайнов описывают форму как единое целое.

Спрайт (sprite) — растровое изображение отдельного объекта рисунка, которое сохраняется в битовом массиве и быстро копируется в нужное место. Спрайты широко используются в анимации.

Тексел (texel — texture element) — элемент растровой текстуры.

Текстура (texture) — стиль закрашивания, который создает иллюзию рельефности поверхности объекта. Часто используется в виде растровых образцов (битмапов).

Трассировка лучей (raytracing) — методы создания реалистичных изображений, основанные на отслеживании распространения световых лучей.

- Обратная *тр. л.* — отслеживание световых лучей в обратном порядке, то есть, начиная от точки наблюдения и далее, к источникам света.

Триангуляция — формирование модели поверхности в виде множества связанных треугольников.

Фильтрация текстур — способ коррекции, интерполяции изображения при наложении текстуры на поверхность объектов.

Фонга метод — способ закрашивания граней трехмерных объектов, который основывается на интерполяции векторов нормалей в вершинах.

Фракталы (fractals) — объекты сложной формы, которые описываются простыми циклами итераций.

Шрифт (font) — набор знаков символов для представления текста в полиграфии, компьютерных системах, причем для этих знаков характерны единство стиля, размеров, одинаковость способов отображения.

- **Растровый *шр.*** — набор растровых изображений символов.
- **Векторный *шр.*** — использует описание символов в векторной форме, благодаря чему может гибко изменять размеры и форму текста.
- **TrueType** — формат шрифтов для программ операционной системы Windows, является разновидностью векторного шрифта — использует описание формы символов B-сплайнами.

AGP (Accelerated Graphics Port) — локальная шина для обмена информацией между видеоадаптером, процессором и оперативной памятью.

API (Application Program Interface) — интерфейс для разработки прикладных программ.

BMP — растровый графический формат файлов, который широко используется программами в операционной среде Windows. Изображение сохраняется в форме битового массива.

CMYK (Cyan Magenta Yellow blac) — субтрактивная цветовая модель.

DirectX — подсистема Windows для графического вывода.

Direct3D — программный интерфейс API, разработанный в Microsoft для трехмерной графики в Windows.

dpi (dots per inch) — количество точек на дюйм длины. Единица измерения разрешающей способности растра.

DXF — векторный графический формат файлов. Разработан Autodesk.

EMF (Enhanced MetaFile) — векторный графический формат для программ в среде Windows.

FIF (Fractal Image Format) — формат файлов для изображений, которые сжаты методом фрактального сжатия.

fps (frames per second) — количество кадров в секунду. Единица измерения скорости видеосистемы для мультимедиа.

GIF — растровый графический формат, который широко используется в Internet. Разработан CompuServe.

GLide — программный интерфейс API, разработанный в 3Dfx Interactive для графических программ, использующих графические видеоадаптеры семейства Voodoo.

HTML — формат файлов для документов, в которых присутствуют текст, графика и другие элементы. Широко используется в Internet.

JPEG (Joint Photographic Experts Group) — стандарт формата файлов для растровых изображений с эффективным сжатием информации.

LZW — метод сжатия информации, который используется, например, в файлах формата GIF.

MIP mapping — сохранение нескольких вариантов текстуры для различных ракурсов, масштабов показа, что обеспечивает улучшение вывода текстурированных поверхностей.

MPEG (Moving Pictures Expert Group) — стандарт для цифрового кодирования компьютерных видеофильмов.

OpenGL (Open Graphic Library) — библиотека графических функций, интерфейс для графических прикладных программ. Разработана Silicon Graphics.

PCX — популярный растровый графический формат. Разработан ZSoft.

PDF (Portable Document Format) — формат файлов электронных документов. Файл может включать текст, графику (растровую и векторную) и прочие данные. Разработан Adobe.

RAM (Random Access Memory) — оперативная память компьютера.

- **VRAM** (Video RAM) — видеопамять, кадровый буфер в видеоадаптерах.

RGB (Red Green Blue) — аддитивная цветовая модель, согласно которой цвет кодируется тремя компонентами — красным, зеленым и синим.

RGBA (Red Green Blue Alpha) — компоненты описания цвета и прозрачности для элементов изображения.

RLE (Run Length Encoding) — метод сжатия информации. Используется, например, в файлах PCX.

TIFF (Tag Image File Format) — растровый графический формат, который используется для обмена графическими данными.

VRML (Virtual Reality Modeling Language) — компьютерный язык описания трехмерных объектов и сцен. Используется в Internet.

Wavelet transform — метод преобразования, который можно использовать для эффективного сжатия изображений. Внедряется AT&T.

Windows — операционная система для компьютеров. Использует графический интерфейс пользователя. Часто применяется в персональных компьютерах типа IBM PC. Разработана Microsoft.

WMF (Windows MetaFile) — векторный графический формат для программ в среде Windows.

XYZ — название цветовой модели, принятой Международной Комиссией по Освещению.

Z-буфер — массив, в котором сохраняются значения расстояния до точки наблюдения (глубина) для каждого пиксела растрового изображения. Используется в алгоритмах создания изображений трехмерных объектов с удалением невидимых точек.

Список литературы

Книги, монографии

1. Ашкенази Г. И. Цвет в природе и технике. — М.: Энергоатомиздат, 1985. — 96 с.
2. Борн Г. Форматы данных. — СПб.: BHV, 1995. — 472 с.
3. Бронштейн И. Н., Семендяев К. А. Справочник по математике. — М., Л.: ОГИЗ, 1948. — 556 с.
4. Вул. Л. Web-графика: справочник. — СПб.: Питер, 1998. — 234 с.
5. Гантмахер Ф. Р. Теория матриц. — М.: Наука, 1967. — 576 с.
6. Григорьев В. Л. Видеосистемы ПК фирмы IBM. — М.: Радио и связь, 1993. — 192 с.
7. Гук М. Процессоры Intel: от 8086 к Pentium II. — СПб.: Питер, 1998. — 224 с.
8. Домбругов Р. М. Телевидение. — Киев: Высшая школа. Головное изд-во, 1979. — 176 с.
9. Иванов В. П., Батраков А. С. Трехмерная компьютерная графика. — М.: Радио и связь, 1995. — 223 с.
10. Ивенс. Р. М. Введение в теорию цвета. — М.: Мир, 1964. — 442 с.
11. Калверт Ч. Borland C++ Builder 3. Энциклопедия пользователя. — Киев: ДиаСофт, 1998. — 800 с.
12. Кенцл Т. Форматы файлов Internet. — СПб.: Питер, 1997. — 320 с.
13. Кошкин Н. И., Ширкевич М. Г. Справочник по элементарной физике. — М.: Наука, 1980. — 208 с.

14. Кузнецов Ю. В., Успенский В. А. Электронное растрирование в полиграфии. — М.: Книга, 1976. — 144 с.

15. Луизов А. В. Цвет и свет. — Ленинград: Энергоатомиздат, 1989. — 256 с.

16. Мусхелишвили Н. И. Курс аналитической геометрии. — М.: Высшая школа, 1967. — 656 с.

17. Нечаева И. А. Цветоведение и теория трехцветной репродукции. — М.: Искусство, 1956. — 197 с.

18. Ньюмен У., Спрулл Р. Основы интерактивной машинной графики. — М.: Мир, 1976.

19. Павлидис Т. Алгоритмы машинной графики и обработки изображений. — М.: Радио и связь, 1986.

20. Петзолд Ч. Программирование для Windows 95. — СПб.: BHV, 1997.

21. Полянский Н. Н. Основы полиграфического производства. — М.: Книга, 1991. — 352 с.

22. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — 512 с.

23. Смирнов В. И. Курс высшей математики. — М., Л.: ОГИЗ — Гостехиздат, 1948.

24. Страуструп Б. Язык программирования C++. В 2-х частях. — Киев: ДиаСофт, 1993.

25. Тихомиров Ю. Программирование трехмерной графики. — СПб.: BHV — Санкт-Петербург, 1998. — 256 с.

26. Томпсон Н. Секреты программирования трехмерной графики для Windows 95. Перев. с англ. — СПб.: Питер, 1997. — 352 с.

27. Федер Е. Фракталы. — М.: Мир, 1991. — 254 с.

28. Фоли. Дж., ван Дэм А. Основы интерактивной машинной графики. В 2-х книгах. — М.: Мир, 1985.

29. Фролов А. В., Фролов Г. А. Программирование видеоадаптеров CGA, EGA и VGA. — М.: ДИАЛОГ-МИФИ, 1992. — 288 с. (Библиотека системного программиста; Т. 3.)

30. Хвольсон О. Д. Курс физики. В пяти томах. Издание пятое. Том второй: — Берлин, Государственное издательство, 1923. — 774 с.

31. Хорн Б. К. П. Зрение роботов. — М.: Мир, 1989. — 487 с.

32. Шикин Е. В., Боресков А. В. Компьютерная графика. — М.: ДИАЛОГ-МИФИ, 1995.

33. Эгрон Ж. Синтез изображений. Базовые алгоритмы. — М.: Радио и связь, 1993.

34. Яншин В. В., Калинин Г. А. Обработка изображений на языке Си для IBM PC: Алгоритмы и программы. — М.: Мир, 1984. — 241 с.

35. Ярмола Ю. А. Компьютерные шрифты. — СПб.: BHV — Санкт-Петербург, 1994. — 208 с.

Статьи в периодических изданиях, доклады на конференциях

36. Ансон Л., Барнсли М. Фрактальное сжатие изображения //Мир ПК, 1992, № 4, с. 52—58.

37. Аненков А. Добро пожаловать в третье измерение //Известия, 28 декабря 1999 г., с. 8.

38. Блинова Т. А., Порев В. Н. Алгоритм определения центров яркости и площадей объектов полутоновых изображений //Тезисы докладов школы-семинара "Обработка изображений в цифровых системах". — Киев: 1992, с. 4—6.

39. Бриджес Дж. Режим X //Журнал д-ра Добба, 1991, № 3, с. 44—46.

40. Вакуленко А. Современные технологии памяти //Компьютерное обозрение, 1998, № 12.

41. Коваленко В. Текстура в задачах трехмерной визуализации // Открытые системы, 1996, № 6.

42. Кустовская Л. Цветовые модели и их цветовой охват //Компьютерное обозрение, 1998, № 27, с. 26—27.

43. Кустовский Д. В борьбе за качеством печати //Компьютерное обозрение, 1998, № 21, с. 12—20.

44. Мангер В. Возможности технологии 3DNow! // Компьютеры+Программы, 1999, № 2, с. 12—15.

45. Митилино С. Я печатаю дом. //Компьютерное обозрение, 2000, № 38, с. 54—55.

46. Моисеенко С. Кратко о фильтрации текстур //Компьютерное обозрение, 1998, № 48, с. 18—23.

47. Молодчик П. Оптические потоки //Компьютерное обозрение, 2000, № 13, с. 40—44.

48. Огарков В. М. От триангуляции Делоне к управляемой триангуляции //Матеріали Четвертої Всеукраїнської конференції з геоінформаційних технологій "Теорія, технологія, впровадження ГІС" ГІС — Форум — 1998. — Київ: 1998, с. 19—24.

49. Помперт А., Пфлессер Б., Риемер М., Шиеманн Т., Шуберт Р., Тиеде В., Хон К. Х. Визуализация объема в медицине //Открытые системы, 1996, № 5.

50. Потапов М. Воксельная графика: великолепная альтернатива // Компьютерное обозрение, 1999, № 40, с. 30—33.

51. Смалий А. Электронная бумага — миф? Отнюдь... //Компьютерное обозрение, 1998, № 33, с. 42—43.

52. Шевченко В. Фракталы в компьютерных технологиях //Компьютерное обозрение, 1997, № 23, с. 24—27.

53. Ягель Р. Рендеринг объемов в реальном времени //Открытые системы, 1996, № 5.

54. Bresenham J. E. Algorithm for computer control of a digital plotter //IBM System Journal, Vol. 4, n. 1, pp. 25—30, 1965.

55. Bresenham J. E. A linear algorithm for incremental digital display of circular arcs //CGIP, Vol. 20, n. 2, 1977.

56. Kaufman A. Volume Visualization //IEEE Computer Society Press, Los Alamitos, CA, 1991.

Документация программных продуктов

57. Геоінформаційна система "ОКО". Керівництво користувача. Книга 3. — Київ: Геобіономіка, 1996.

58. 3D Studio MAX R3. Справочная документация. Autodesk Inc., 1999.

59. Borland C++ 5.02. Programming guide. Borland International. 1997.

60. Bryce 4. User Guide. MetaCreations Corporation. 1999.

61. Software Development Kit (SDK). Interface Win32: Microsoft Corporation, 1996 (и более поздние издания).

Сеть Internet

Много разнообразной информации различного качества можно получить в сети Internet. По компьютерной графике существуют тысячи сайтов. Отыскать необходимые сведения возможно с помощью поисковых серверов, таких как **www.altavista.com**, **www.yahoo.com**, **www.rambler.ru** и других.