Андре Ламот
Tricks of the Windows Game Programming Gurus Second Edition
Andre LaMothe

Книга предназначена для читателей, интересующихся вопросами разработки игр в операционной системе Windows. В ней освещены разнообразные аспекты программирования игр — от азов программирования до серьезного рассмотрения различных компонентов DirectX, от простейших физических моделей до сложных вопросов искусственного интеллекта. Книга будет полезна как начинающим, так и профессиональным разработчикам игр для Windows, хотя определенные знания в обл программирования (в частности, языка программирования C или C++), математики и физики существенно облегчат изучение материала.

ББК 32.973.26-018.2.75
Л21
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *С.В. Беликовой*, канд. техн. наук. *И.В. Красикова*, *А.И. Мороза*, *В.Н. Романова*

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

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

Ламот, Андре.

Л21 Программирование игр для Windows. Советы профессионала, 2-е изд. : Пер. с англ. — М. : Издательский дом "Вильямс", 2003. — 880 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0422-6 (рус.)

Книга предназначена для читателей, интересующихся вопросами разработки игр для операционной системы Windows. В ней освещены разнообразные аспекты — от азов программирования до серьезного рассмотрения различных компонентов DirectX, от простейших физических моделей до сложных вопросов искусственного интеллекта. Книга будет полезна как начинающим, так и профессиональным разработчикам игр для Windows, хотя определенные знания в области программирования (в частности, языка программирования C или C++), математики и физики существенно облегчат изучение материала.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2002
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2003

ISBN 5-8459-0422-6 (рус.)
ISBN 0-672-32369-9 (англ.)

© Издательский дом "Вильямс", 2003
© Sams Publishing, 2002

Оглавление

Предисловие 17
Об авторе 19
Благодарности 20
Введение 21

Часть I. Основы программирования в Windows 27
Глава 1. Путешествие в пропасть 29
Глава 2. Модель программирования Windows 61
Глава 3. Профессиональное программирование в Windows 99
Глава 4. GDI, управляющие элементы и прочее 157

Часть II. DirectX и основы двухмерной графики 193
Глава 5. Основы DirectX и COM 195
Глава 6. Первое знакомство с DirectDraw 219
Глава 7. Постигаем секреты DirectDraw и растровой графики 257
Глава 8. Растеризация векторов и двухмерные преобразования 365
Глава 9. DirectInput 499
Глава 10. DirectSound и DirectMusic 541

Часть III. Программирование игр 591
Глава 11. Алгоритмы, структуры данных, управление памятью и многопоточность 593
Глава 12. Искусственный интеллект в игровых программах 647
Глава 13. Основы физического моделирования 707
Глава 14. Генерация текста 765
Глава 15. Разработка игры Outpost 801

Часть IV. Приложения 825
Приложение А. Содержание компакт-диска 827
Приложение Б. Установка DirectX и использование компилятора C/C++ 829
Приложение В. Обзор математики и тригонометрии 833
Приложение Г. Азы C++ 845
Приложение Д. Ресурсы по программированию игр 865
Предметный указатель 869

Содержание

Предисловие 17

Об авторе 19

Благодарности 20

Введение 21

Часть I. Основы программирования в Windows 27

Глава 1. Путешествие в пропасть 29

Небольшая история 29
Проектирование игр 31
Типы игр 31
Мозговой штурм 33
Разработка документации и сценария 33
Жизнеспособность игры 34
Компоненты игры 34
- Инициализация 34
- Вход в цикл игры 34
- Получение пользовательского ввода 35
- ИИ и логика игры 35
- Подготовка очередного кадра 35
- Синхронизация вывода 35
- Цикл 36
- Завершение работы 36

Общие рекомендации по программированию игр 38
Использование инструментов 42
- Компиляторы C/C++ 42
- Программное обеспечение для двухмерной графики 42
- Программное обеспечение для обработки звука 42
- Моделирование трехмерных объектов 42
- Музыкальные и MIDI-программы 43

Использование компилятора 43
Пример: игра FreakOut 44
Резюме 60

Глава 2. Модель программирования Windows 61

Происхождение Windows 61
- Ранние версии Windows 62
- Windows 3.x 62
- Windows 95 62
- Windows 98 63
- Windows ME 63
- Windows XP 63

Windows NT/2000 64
Базовая архитектура Windows: Win9X/NT 64
Многозадачность и многопоточность 64
Получение информации о потоках 65
Модель событий 67
Венгерская нотация 68
Именование переменных 69
Именование функций 69
Именование типов и констант 69
Именование классов 69
Именование параметров 70
Простейшая программа Windows 70
Все начинается с WinMain() 71
Разбор программы 72
Выбор окна сообщения 74
Написание реального Windows-приложения 76
Класс Windows 77
Регистрация класса Windows 83
Создание окна 83
Обработчик событий 86
Главный цикл событий 91
Цикл сообщений для работы в реальном времени 95
Открытие нескольких окон 96
Резюме 98

Глава 3. Профессиональное программирование в Windows 99

Использование ресурсов 99
Размещение и работа с ресурсами 101
Использование ресурсов пиктограмм 102
Использование ресурсов курсоров 105
Создание ресурсов таблицы строк 109
Использование звуковых ресурсов 110
Использование компилятора для создания .RC-файлов 115
Работа с меню 116
Создание меню 116
Загрузка меню 118
Обработка сообщений меню 121
Введение в GDI 126
Еще раз о сообщении WM_PAINT 126
Основы видеовывода и цвета 130
Видеорежимы RGB и палитры 132
Вывод текста 133
Обработка основных событий 139
Управление окном 139
Работа с клавиатурой 145
Работа с мышью 151
Отправление сообщений самому себе 154
Резюме 156

Глава 4. GDI, управляющие элементы и прочее **157**
Работа с GDI 157
Контекст графического устройства 157
Цвет, перья и кисти 158
Работа с перьями 159
Работа с кистями 162
Точки, линии, многоугольники и окружности 163
Точка 163
Линия 165
Прямоугольник 166
Окружность и эллипс 168
Многоугольник 170
Текст и шрифты 171
Таймер 172
Сообщение WM_TIMER 173
Низкоуровневая работа со временем 175
Управляющие элементы 178
Кнопки 178
Отправка сообщений дочерним управляющим элементам 181
Получение информации 183
T3D Game Console 187
Резюме 191

Часть II. DirectX и основы двухмерной графики **193**

Глава 5. Основы DirectX и COM **195**
Азы DirectX 195
HEL и HAL 198
Подробнее о базовых классах DirectX 198
COM: дело рук Microsoft... или врага рода человеческого? 200
Что такое COM-объект 201
Еще об идентификаторах интерфейсов и GUID 204
Создание COM-объекта 205
Краткое повторение COM 207
Пример COM-программы 207
Работа с COM-объектами DirectX 211
COM и указатели на функции 212
Создание и использование интерфейсов DirectX 215
Запрос интерфейсов 216
Будущее COM 218
Резюме 218

Глава 6. Первое знакомство с DirectDraw **219**
Интерфейсы DirectDraw 219
Характеристики интерфейсов 220
Совместное использование интерфейсов 222
Создание объекта DirectDraw 223
Обработка ошибок в DirectDraw 224
Обновление версии интерфейса 225
Взаимодействие с Windows 227

Установка режима 231
О тонкостях работы с цветом 235
Построение поверхности дисплея 239
Создание первичной поверхности 240
Присоединение палитры 247
Вывод пикселей на экран 247
Очистка 255
Резюме 256
Глава 7. Постигаем секреты DirectDraw и растровой графики 257
Высокоцветные режимы 257
Шестнадцатибитовый высокоцветный режим 258
Получение информации о формате пикселей 260
Высокоцветный 24/32-битовый режим 268
Двойная буферизация 271
Динамика поверхностей 276
Переключение страниц 279
Использование блиттера 286
Заполнение памяти с помощью блиттера 288
Копирование изображений из одной поверхности в другую 295
Основные принципы работы с отсечением 299
Отсечение пикселей, выходящих за пределы области видимости 299
Трудоемкий способ отсечения битовых образов 301
Отсечение с помощью интерфейса IDirectDrawClipper 306
Работа с битовыми образами 312
Загрузка файлов в формате .BMP 312
Использование битовых образов 319
Загрузка 8-битовых образов 320
Загрузка 16-битовых образов 321
Загрузка 24-битовых образов 322
Заключительное слово по поводу битовых образов 323
Внеэкранные поверхности 323
Создание внеэкранных поверхностей 323
Блиттинг внеэкранных поверхностей 325
Настройка блиттера 326
Цветовые ключи 326
Выбор цветовых ключей источника 328
Выбор цветовых ключей поверхности назначения 330
Использование блиттера (наконец-то!) 331
Вращение и масштабирование битовых образов 333
Теория дискретизации 335
Цветовые эффекты 339
Цветовая анимация в 256-цветных режимах 339
Циклическая перестановка цветов в 256-цветном режиме 345
Приемы, применяющиеся в режимах RGB 347
Преобразование цветов вручную и таблицы цветов 347
Новые интерфейсы DirectX 348
Совместное использование GDI и DirectX 349
Вглубь DirectDraw 351
Основной объект DirectDraw 351

Получение информации о поверхностях 353
Получение информации о палитрах 353
DirectDraw и оконные режимы 354
Вывод пикселей в окне 356
Определение параметров реальной клиентской области 359
Отсечение в окнах DirectX 361
Работа в 8-битовых оконных режимах 362
Резюме 363
Глава 8. Растеризация векторов и двухмерные преобразования 365
Черчение прямых линий 365
Алгоритм Брезенхама 367
Повышение скорости алгоритма 373
Основы отсечения двухмерных изображений 374
Вычисление точки пересечения с использованием одной точки и наклона 376
Канонический вид уравнения прямой 378
Вычисление пересечения двух линий с помощью матриц 378
Отсечение линии 380
Алгоритм Кохена–Сазерленда 381
Каркасные многоугольники 387
Структуры данных многоугольников 388
Черчение и отсечение многоугольников 390
Преобразования, выполняемые на плоскости 392
Перенос 392
Поворот 394
Введение в матрицы 403
Единичная матрица 405
Сложение матриц 405
Умножение матриц 406
Преобразования, выполняемые с помощью матриц 408
Перенос 409
Масштабирование 409
Поворот 410
Сплошные заполненные многоугольники 412
Типы треугольников и четырехугольников 413
Черчение треугольников и четырехугольников 414
Подробности процесса разделения треугольника 417
Общий случай растеризации четырехугольника 425
Триангуляция многоугольников 426
Обнаружение столкновений при работе с многоугольниками 430
Описанная окружность/сфера 430
Ограничивающий прямоугольник 433
Содержание точки 435
Немного о хронометрировании и синхронизации 437
Прокрутка и панорамирование 439
Подсистемы прокрутки страниц 439
Однородные подсистемы элементов мозаичного изображения 440
Подсистемы разреженных растровых мозаичных изображений 445
Псевдотрехмерные изометрические подсистемы 446
Метод 1 447

Метод 2 448
Метод 3 450
Библиотека T3DLIB1 450
Архитектура графической подсистемы 451
T3DCONSOLE2.CPP: новая консоль для программирования игр 451
Основные определения 459
Рабочие макросы 460
Структуры и типы данных 461
Глобальные переменные 464
Интерфейс DirectDraw 466
Функции для работы с многоугольниками 470
Двухмерные графические примитивы 473
Математические функции и функции ошибок 477
Функции для работы с растровыми изображениями 479
Функции для работы с палитрой 484
Служебные функции 487
Работа с объектами блиттера 489
Резюме 498

Глава 9. DirectInput 499

Цикл ввода: повторение пройденного 499
DirectInput: вступительная часть 501
Компоненты системы DirectInput 502
Настройка DirectInput 503
Режимы получения данных 505
Создание основного объекта DirectInput 505
Клавиатура со 101 клавишей 506
Повторный захват устройства 514
Мышеловка 515
Работа с джойстиком 519
Библиотека системы обобщенного ввода T3DLIB2.CPP 534
Краткий обзор библиотеки T3D 539
Резюме 540

Глава 10. DirectSound и DirectMusic 541

Программирование звука на компьютере 541
А затем был звук... 542
Цифровой звук и MIDI 545
Цифровой звук 546
Синтезированный звук и MIDI 547
MIDI 548
Аппаратное звуковое обеспечение 549
Табличный синтез 549
Волноводный синтез 550
Цифровая запись: инструменты и технологии 550
Запись звуков 551
Обработка звуков 551
DirectSound 552
Запуск системы DirectSound 554
Понятие уровня взаимодействия 555

Задание уровня взаимодействия 555
Основной и дополнительный аудиобуферы 556
Работа с дополнительными буферами 557
Создание дополнительных аудиобуферов 558
Запись данных во вторичные буферы 560
Воспроизведение звука 562
Управление звуком 562
Изменение частоты 563
Настройка стереобаланса 564
Общение с DirectSound 564
Чтение звуков с диска 566
Формат .WAV 566
Чтение .WAV-файлов 566
Большой эксперимент DirectMusic 571
Архитектура DirectMusic 572
Запуск DirectMusic 573
Инициализация COM 573
Создание исполняющей системы 574
Добавление порта 575
Загрузка MIDI 575
Создание загрузчика 575
Загрузка MIDI-файла 576
Работа с MIDI-сегментами 578
Воспроизведение MIDI-сегмента 578
Остановка воспроизведения MIDI-сегмента 579
Проверка состояния MIDI-сегмента 579
Освобождение MIDI-сегмента 579
Завершение работы с DirectMusic 579
Пример использования DirectMusic 580
Звуковая библиотека T3DLIB3 580
Заголовочный файл 580
Типы 581
Глобальные переменные 582
DirectSound API 582
DirectMusic API 587
Резюме 589

Часть III. Программирование игр 591

Глава 11. Алгоритмы, структуры данных, управление памятью и многопоточность 593

Структуры данных 593
Статические структуры и массивы 594
Связанные списки 594
Анализ алгоритмов 600
Рекурсия 602
Деревья 603
Построение бинарных деревьев 607
Поиск в бинарном дереве 609
Теория оптимизации 611

Работайте головой 611
Математические уловки 612
Математические вычисления с фиксированной точкой 613
Развертывание цикла 617
Таблицы поиска 617
Язык ассемблера 618
Создание демоверсий 619
Предварительно записанная версия 619
Демоверсия с использованием ИИ 620
Стратегии сохранения игр 621
Реализация игр для нескольких игроков 621
Поочередная игра 621
Разделение экрана 622
Многопоточное программирование 623
Терминология многопоточного программирования 624
Зачем нужны потоки в играх? 626
Создание потоков 627
Пересылка сообщений между потоками 634
Ожидание завершения потоков 638
Многопоточность и DirectX 645
Дополнительные вопросы многопоточности 646
Резюме 646

Глава 12. Искусственный интеллект в игровых программах 647

Азы искусственного интеллекта 647
Детерминированные алгоритмы искусственного интеллекта 648
Случайное движение 649
Алгоритм следования за объектом 650
Алгоритм уклонения от объекта 654
Шаблоны и сценарии 655
Шаблоны 655
Шаблоны с обработкой условий 659
Моделирование систем с состояниями 660
Элементарные конечные автоматы 662
Добавление индивидуальности 665
Запоминание и обучение 668
Деревья планирования и принятия решений 669
Кодирование планов 671
Реализация реальных планировщиков 675
Поиск пути 676
Метод проб и ошибок 676
Обход по контуру 678
Избежание столкновений 678
Поиск путей с использованием промежуточных пунктов 679
Гонки 681
Надежный поиск пути 682
Сценарии в работе искусственного интеллекта 687
Разработка языка сценариев 687
Использование компилятора C/C++ 689
Искусственные нейронные сети 694

Генетические алгоритмы 696
Нечеткая логика 698
Теория обычных множеств 699
Теория нечетких множеств 700
Переменные и правила нечеткой лингвистики 701
Нечеткая ассоциативная матрица 703
Создание реальных систем ИИ для игр 704
Резюме 705
Глава 13. Основы физического моделирования 707
Фундаментальные законы физики 708
Масса (m) 708
Время (t) 709
Положение в пространстве (s) 709
Скорость (v) 711
Ускорение (a) 712
Сила (F) 715
Силы в многомерном пространстве 715
Импульс (P) 716
Законы сохранения 717
Гравитация 718
Моделирование гравитации 720
Моделирование траекторий снарядов 722
Трение 724
Основы трения 724
Трение на наклонной плоскости 726
Столкновения 729
Простой удар 729
Столкновение с плоскостью произвольной ориентации 731
Пример отражения вектора 734
Пересечение отрезков 734
Реальное двухмерное столкновение объектов 739
Система координат n-t 743
Простая кинематика 749
Решение прямой задачи кинематики 750
Решение обратной задачи кинематики 752
Системы частиц 753
Что требуется для каждой частицы 753
Разработка процессора частиц 754
Программный процессор частиц 755
Генерация начальных условий 759
Построение физических моделей игр 762
Структуры данных для физического моделирования 762
Моделирование на основе кадров и на основе времени 762
Резюме 764
Глава 14. Генерация текста 765
Что такое текстовая игра 766
Как работает текстовая игра 767
Получение входных данных из внешнего мира 770

Анализ и синтаксический разбор языка 771
Лексический анализ 776
Синтаксический анализ 781
Семантический анализ 784
Разработка игры 784
Представление мира игры 785
Размещение объектов в мире игры 787
Выполнение действий 788
Перемещения 788
Система инвентаризации 789
Реализация обзора, звуков и запахов 789
Запах 791
Обзор 792
Игра в реальном времени 794
Обработка ошибок 795
Игра “Земля теней” 795
Язык “Земли теней” 796
Построение “Земли теней” и игра в нее 797
Цикл игры “Земля теней” 798
Победитель игры 799
Резюме 799

Глава 15. Разработка игры Outpost 801

Начальный эскиз игры 801
Сюжет игры 802
Проектирование хода игры 803
Инструменты, используемые при написании игры 803
Мир игры: прокрутка пространства 804
Космический корабль игрока 805
Поле астероидов 808
Противники 810
Аванпосты 810
Мины 811
Боевые корабли 813
Запасы энергии 816
Пилотажный дисплей 817
Система частиц 820
Процесс игры 821
Компиляция игры *Outpost* 821
Компилируемые файлы 822
Файлы времени исполнения 822
Эпилог 822

Часть IV. Приложения 825

Приложение А. Содержание компакт-диска 827

Приложение Б. Установка DirectX и использование компилятора C/C++ 829

Использование компилятора C/C++ 830

Приложение В. Обзор математики и тригонометрии **833**
Тригонометрия 833
Векторы 836
Длина вектора 837
Нормирование 837
Умножение на скаляр 837
Сложение векторов 838
Вычитание векторов 839
Внутреннее (скалярное) произведение векторов 839
Векторное произведение 840
Нулевой вектор 842
Радиус-вектор 842
Векторы как линейные комбинации 842
Приложение Г. Азы C++ **845**
Язык программирования C++ 845
Минимум, который необходимо знать о C++ 847
Новые типы, ключевые слова и обозначения 848
Комментарии 848
Константы 848
Ссылки 849
Создание переменных “на лету” 849
Управление памятью 850
Потоки ввода-вывода 851
Классы 853
Структуры в C++ 853
Простой класс 853
Открытые и закрытые члены класса 854
Функции-члены класса 855
Конструкторы и деструкторы 856
Оператор разрешения области видимости 860
Перегрузка функций и операторов 861
Резюме 863
Приложение Д. Ресурсы по программированию игр **865**
Web-узлы по программированию игр 865
Откуда можно загрузить информацию 865
2D/3D-процессоры 866
Книги по программированию игр 866
Microsoft DirectX 866
Конференции Usenet 867
Blues News 867
Журналы, посвященные разработке игр 867
Разработчики игровых Web-узлов 867
Xtreme Games LLC 868
Предметный указатель **869**

Предисловие

Первую книгу Андре Ламота — *Sams Teach Yourself Game Programming in 21 Days* — я купил в 1994 году. До этого мне не встречались люди, которые могли сделать карьеру на программировании видеоигр. И вот я увидел, что между моей любовью к программированию и страстью к видеоиграм может быть непосредственная связь! Кто бы мог подумать, что часы, проведенные за игрой на компьютере, могут рассматриваться как исследовательская работа? Книга Андре заставила меня поверить в собственные силы и в то, что я могу создавать игры. Я позвонил ему (до сих пор не могу поверить, что он рискнул привести в книге свой телефонный номер и отвечать на все звонки читателей!) и попросил помощи в работе над простой игровой программой, которую я делал на уроках физики и которая была основана на моделировании поведения газа (но, к сожалению, я никак не мог заставить ее работать). Он просмотрел мою программу и сказал: "Рич, ты просто убиваешь меня! поставь точки с запятыми в конце каждой строки!" Как оказалось, этого было достаточно для того, чтобы программа заработала.

Несколькими годами позже мне пришлось участвовать в работе над игрой Rex Blade вместе с Андре, в задачи которого входила разработка инструментария и уровней игры. Эта работа необычайно обогатила мой опыт и многому научила меня. Она была чрезвычайно сложной (Андре — еще тот надсмотрщик и погоняла), тем не менее доставляла огромное удовольствие и в конечном счете вылилась в трехмерную интерактивную игровую видеотрилогию. Вся работа над игрой — от идеи до ее появления на магазинной полке — заняла шесть незабываемых месяцев и сделала меня настоящим программистом в области игр. Я понял, что такое настоящая работа над видеоиграми и как нужно уметь работать, а ведь сначала я просто не верил Андре, когда он говорил, что ему приходилось работать и по 100 часов в неделю.

Существует не так уж много областей, где программист должен быть мастером на все руки, и программирование игр — одна из них. Игра состоит из множества разнородных частей, которые должны идеально работать вместе: графика, звуковые эффекты, музыка, графический интерфейс, система искусственного интеллекта и многое другое. При работе над играми необходимо не только владеть искусством программирования и уметь правильно выбрать алгоритмы и структуры данных, но также знать физику и математику... Обо всем этом можно прочитать в книге, которую вы держите в руках и в которой описан сегодняшний и даже завтрашний день видеоигр.

Эта книга знакомит вас с технологией программирования игр на качественно новом уровне. Достаточно лишь упомянуть об одной теме данной книги — разработке системы искусственного интеллекта, чтобы стал понятен уровень изложения материала. Это книга для профессионалов или тех, кто твердо намерен ими стать, — где еще можно прочитать о таких вещах, как использование в видеоиграх нечеткой логики, нейронных сетей, генетических алгоритмов? Кроме того, книга знакомит вас со всеми важными компонентами DirectX, такими, как DirectDraw, DirectInput, DirectSound и последней разработкой в этой области — DirectMusic.

Немало материала книги посвящено физическому моделированию, в частности вопросам обработки столкновений, законам сохранения, моделированию сил гравитации и трения и тому, как заставить все это работать в режиме реального времени. Представьте себе игру с персонажами, которые могут обучаться, объектами, которые могут весьма натурально сталкиваться друг с другом, врагами, которые запоминают, какую тактику вы применяли в борьбе с ними... Все это описано в книге и служит основой завтрашних игр.

Надо отдать должное Андре за написание этой книги. Он всегда говорил: если не я, то кто же? И это справедливо — книга, написанная тем, у кого за плечами более 20 лет практического опыта в написании игр, действительно в состоянии оказать неоценимую помощь другим.

Меня волнует мысль, что вместе со мной эту книгу читают те, кто будет раздвигать границы возможностей компьютерных игр в 21 веке, что она вдохновит того, кому предстоит работать с аппаратным обеспечением, возможности которого мы еще просто не в состоянии представить. И что кто-то из читателей этой книги придет на смену Андре и когда-нибудь напишет свою собственную книгу.

Ричард Бенсон (Richard Benson),
программист в области трехмерных игр
Dream Works Interactive/Electronics Arts

Об авторе

Андре Ламот (André LaMothe, известный в определенных кругах как Lord Necron) имеет более чем 24-летний стаж программиста и дипломы в области математики, информатики и электроники. Им написан ряд статей на различные темы, включая такие, как графика, программирование игр и искусственный интеллект. Его перу принадлежат такие бестселлеры, как *Tricks of the Game Programming Gurus*, *Sams Teach Yourself Game Programming in 21 Days*, *The Game Programming Starter Kit*, *The Black Art of 3D Game Programming* и *Windows Game Programming for Dummies*. Кроме того, Ламч от преподает на кафедре расширений мультимедиа в университете Санта-Круз.

Написать ему можно по адресу: ceo@xgames3d.com.

Я посвящаю эту книгу невинным людям,
пострадавшим от преступлений

Благодарности

Я всегда ненавидел писать благодарности, поскольку в процессе создания книги участвует слишком много людей, чтобы всех их можно было поблагодарить. Тем не менее я попытаюсь это сделать.

И в первую очередь я хочу поблагодарить родителей за то, что они дали мне жизнь и набор генов, который позволяет мне почти не спать и непрерывно работать. Спасибо, родные мои!

Затем я хотел бы выразить признательность всем сотрудникам Sams Publishing за то, что они позволили мне написать эту книгу так, как я хотел, что довольно необычно для корпоративной Америки. В частности, главному редактору Киму Спилкеру (Kim Spilker) за то, что он прислушался к моему мнению о концепции книги; редактору проекта Джорджу Недефу (George Nedeff), который всерьез воспринял мою просьбу о минимальном редактировании книги; специалисту по носителям информации Дану Шерфу (Dan Scherf), который помог собрать прилагаемый к книге компакт-диск; художественному редактору Марку Ренфроу (Mark Renfrow), чьи усилия привели к тому, что сотни рисунков в книге выглядят корректно; и наконец, техническому редактору Дэвиду Франсону (David Franson). Спасибо всем тем, кто работал над этой книгой.

Я бы хотел также поблагодарить группу разработчиков DirectX из Microsoft, в первую очередь Кевина Бахуса (Kevin Bachus) и Стаси Цурусаки (Stacey Tsurusaki) за последние версии DirectX SDK, а также за приглашения на все важные встречи, посвященные DirectX.

Я благодарю все компании, продуктами которых пользовался при подготовке книги, а именно: Caligari Corporation за TrueSpace, JASC за Paint Shop Pro, Sonic Foundry за Sound Forge. Кроме того, я признателен Matrox и Diamond Multimedia за демонстрационные программы к 3D-ускорителям, Creative Labs за звуковые карты, Intel Corporation за VTune, Kinetics за 3D Studio Max и Microsoft® и Borland® за их компиляторы.

Мне бы хотелось сказать сердечное спасибо всем моим друзьям, которые не покидали меня во время этой жуткой работы над книгой. Это парни из Shamrock's Universal Submission Academy: Чокнутый Боб, Джош, Келли, Жавье, Большой Папа, Жидкий Роб, Брайен, Кукла и все остальные. Спасибо Майку Перону, который всегда находил те программы, которые были мне нужны, и Счастливчику Марку (ты не забыл, что уже восемь лет должен мне 180 баксов?).

Еще хочу поблагодарить всех тех, кто любезно позволил разместить свои статьи на прилагаемом компакт-диске, и в первую очередь Мэтью Эллиса (Matthew Ellis), а также Ричарда Бенсона (Richard Benson) за его предисловие к книге.

Спасибо всем!

Введение

Однажды я написал книгу о программировании игр — *Tricks of the Game Programming Gurus*. Для меня это была возможность осуществить давно задуманное и поделиться с читателем своим опытом в написании игр. С тех пор прошло несколько лет, я стал старше и умнее, многому научился и вновь готов поделиться этим с читателями.

Как обычно, я не рассчитываю на то, что мой читатель — опытный программист, а кроме того, имеет навыки написания игр. Эта книга будет полезна начинающему программисту в той же степени, что и профессионалу.

Сегодня, пожалуй, наиболее подходящее время для того, чтобы заняться созданием игр. Уже сейчас существуют игры, удивительно похожие на реальность, а потому можете себе представить, что же нас ждет завтра! Но современные технологии далеко не просты и требуют массу усилий и кропотливой работы. Уровень профессионализма, необходимого для разработки игр, неуклонно растет, но, внимательно прочитав эту книгу, вы, пожалуй, будете в состоянии принять этот вызов.

О чем эта книга

В книге содержится огромное количество информации, которой я хочу до отказа заполнить вашу нейронную сеть. Если говорить серьезно, то здесь вы найдете все необходимое для того, чтобы создать игру для персонального компьютера под управлением операционных систем Windows 9X/NT/XP/2000.

- Программирование Win32.
- Основы DirectX.
- Двухмерная графика и алгоритмы.
- Методы программирования игр и структуры данных.
- Многопоточное программирование.
- Искусственный интеллект.
- Физическое моделирование.
- Использование аппаратного ускорения трехмерной графики (на прилагаемом компакт-диске).

И это далеко не все...

Что вы должны знать

Чтение этой книги предполагает знакомство с программированием. Вам не стоит браться за книгу, если вас приводят в ужас исходные тексты программ на языке C. В данной книге используется C++, но не более чем в режиме “улучшенного C”; кроме того, в книге имеется приложение, кратко описывающее отличия C++ от C, так что, зная C, вы вполне сможете читать эту книгу. Знание C++ в основном требуется для работы с примерами с использованием этого языка.

Главный вывод из сказанного: если вы знакомы с языком C, все в порядке. Если вы знакомы с C++, вам вовсе не о чем беспокоиться.

Все знают, что компьютерная программа представляет собой сплав логики и математики. Создание видеоигр, особенно трехмерных, — это сплошная математика! Однако, по сути, все, что вам нужно из нее знать, — это азы алгебры и геометрии. Что такое век-

торы и матрицы и как с ними работать, вы узнаете из книги, так что если вас научили умножать и делить, то вы должны понять как минимум 90% написанного здесь.

Как организована эта книга

Книга разделена на 4 части, содержит 15 глав и 5 приложений.

Часть I. Основы программирования в Windows

Глава 1. Путешествие в пропасть
Глава 2. Модель программирования Windows
Глава 3. Профессиональное программирование в Windows
Глава 4. GDI, управляющие элементы и прочее

Часть II. DirectX и основы двухмерной графики

Глава 5. Основы DirectX и COM
Глава 6. Первое знакомство с DirectDraw
Глава 7. Постигаем секреты DirectDraw и растровой графики
Глава 8. Растеризация векторов и двухмерные преобразованияс
Глава 9. DirectInput
Глава 10. DirectSound и DirectMusic

Часть III. Программирование игр

Глава 11. Алгоритмы, структуры данных, управление памятью и многопоточность
Глава 12. Искусственный интеллект в игровых программах
Глава 13. Основы физического моделирования
Глава 14. Генерация текста
Глава 15. Разработка игры Outpost

Часть IV. Приложения

Приложение А. Содержание компакт-диска
Приложение Б. Установка DirectX и использование компилятора C/C++
Приложение В. Обзор математики и тригонометрии
Приложение Г. Азы C++
Приложение Д. Ресурсы по программированию игр

Прилагаемый компакт-диск

На прилагаемом компакт-диске находятся все исходные тексты, выполнимые файлы, демонстрационные программы, дополнительные технические статьи и прочие материалы. Вот структура каталогов компакт-диска:

```
CD-DRIVE:\
  T3DGameR1\
    Applications\
      ACROBAT\
      MSWordView\
      PaintShopPro\
      SForge\
      TrueSpace\
      WINZIP\
    Articles\
      3DTechSeries\
```

```
    3DViewing\
    AIandBeyond\
    AppliedAI\
    ArtificialPersonalities\
    ArtOfLensFlares\
    D3DTransparency\
    DesigningThePuzzle\
    Dynamic3DAnimation\
    GPMegaSiteArticles\
    HumanCharacters\
    IntoTheGreyzone\
    KDTrees\
    LinkingUpDirectPlay\
    MusicalContentFundamentals\
    Netware\
    NetworkTrafficReduction\
    OptimizingWithMMX\
    PentiumSecrets\
    PolySorting\
    SmallGroupGameDev\
    TextureMapMania\
    TileGraphics\
    WebGamesOnAShoeString\
    XtremeDebugging\
  Artwork\
    3D Models\
    Bitmaps\
  DirectX\
    DirectXSDK\
  Engines\
    Genesis3D\
    PowerRender\
  Games\
  Onlinebooks\
    Direct3D_Online\
    General3D_Online\
  Sound\
    Midi\
    Waves\
  Source\
    T3DCHAP01\
    T3DCHAP02\
    .....
    T3DCHAP15\
```

Каждый подкаталог содержит определенный раздел с информацией. Приведу более детальное описание некоторых из них.

T3DGameR1 — корневой каталог, который содержит все остальные подкаталоги. Не забудьте прочитать находящийся в нем файл README.TXT, содержащий информацию о последних изменениях.

Source — содержит подкаталоги с исходными текстами, встречающимися в книге. Перенесите этот каталог со всеми подкаталогами на свой жесткий диск и работайте с исходными текстами оттуда.

DirectX — содержит последнюю на момент выхода оригинального издания версию DirectX SDK.

Articles — содержит различные статьи, написанные экспертами в области программирования игр.

Onlinebooks — содержит электронные версии книг, посвященных трехмерной графике.

Установка DirectX

Пожалуй, самое важное, что содержится на прилагаемом компакт-диске, — это DirectX SDK. Программа инсталляции DirectX SDK находится в каталоге DIRECTX\.

НА ЗАМЕТКУ

Для работы с программами на прилагаемом компакт-диске вы должны установить DirectX SDK версии не ниже 8.0. Если вы не уверены в номере установленной у вас версии, запустите программу установки — она сообщит, какая именно версия DirectX установлена в настоящее время в вашей системе.

Компиляция программ

При работе над книгой я использовал компилятор Microsoft Visual C++ 5.0/6.0. Однако в большинстве случаев программы должны корректно компилироваться любым Win32-совместимым компилятором. Тем не менее я считаю, что для работы над играми для Windows лучшим компилятором является Microsoft Visual C++.

Если вы не очень хорошо знакомы с интегрированной средой вашего компилятора, потратьте время на знакомство с ней до чтения книги. У вас не должно возникать вопросов о том, как скомпилировать простейшую программу типа “Hello, world” с помощью компилятора.

Для того чтобы скомпилировать Win32 .EXE-программу, вы должны указать вашему компилятору, что целевая программа — Win32-приложение. При компиляции программ, использующих DirectX, вы также должны вручную включить в ваш проект библиотеки DirectX. В большинстве случаев вам понадобятся следующие библиотеки:

DDRAW.LIB	Импортируемая библиотека DirectDraw
DINPUT.LIB	Импортируемая библиотека DirectInput
DINPUT8.LIB	Импортируемая библиотека DirectInput8
DSOUND.LIB	Импортируемая библиотека DirectSound
D2DIM.LIB	Импортируемая библиотека Direct3D Immediate Mode
DXGUID.LIB	Библиотека DirectX GUID
WINMM.LIB	Расширения мультимедиа Windows

Более детально с правилами компиляции вы познакомитесь при чтении книги. Однако, если вы получите от компилятора сообщение “unresolved symbol”, не спешите слать мне гневные письма — сначала убедитесь, что вы включили в проект все необходимые библиотеки.

Не забудьте также и о том, что, кроме включения в проект библиотечных файлов, вы должны включить заголовочные .h-файлы DirectX и добавить путь к ним в список каталогов поиска заголовочных файлов компилятора. Убедитесь также, что этот путь расположен в списке первым, чтобы нейтрализовать отрицательные последствия возможного включения в состав поставки компилятора старых версий файлов DirectX.

И наконец, если вы используете продукты Borland, то должны обратиться к соответствующим версиям библиотечных файлов DirectX, которые находятся в каталоге BORLAND\ в пакете DirectX SDK.

О втором издании

Второе издание книги существенно дополнено новым материалом. Обновления и дополнения коснулись самых разных тем, в первую очередь последних (весьма существенных!) изменений в DirectX. В этом издании более подробно рассмотрен вопрос о 16-битовых RGB-режимах, при компиляции использовалась версия DirectX 8.0; добавлена новая глава о разборе текста. В целом содержание книги стало более ясным и более организованным.

Не следует забывать, однако, что основная направленность книги — программирование игр, а значит, весь излагаемый материал подчинен этой цели. Именно поэтому книгу ни в коей мере нельзя рассматривать как учебник или сколь-нибудь полное пособие по DirectX или любой другой рассматриваемой в книге теме. Моя основная задача состояла в том, чтобы обучить читателя основам работы с той или иной технологией, познакомить его с областью возможных применений технологии — но при этом постараться, чтобы изложение было максимально простым. Так, например, Direct3D при всей его привлекательности не рассматривается в книге, поскольку этот компонент DirectX довольно сложен для пояснения, а кроме того, он слишком мощный, чтобы использовать его при разработке двухмерных игр.

ЧАСТЬ I

Основы программирования в Windows

Глава 1

Путешествие в пропасть 29

Глава 2

Модель программирования Windows 61

Глава 3

Профессиональное программирование в Windows 99

Глава 4

GDI, управляющие элементы и прочее 157

ГЛАВА 1

Путешествие в пропасть

Программирование в Windows — это непрекращающаяся война с длинной историей. Разработчики игр долгое время воздерживались от работы под Windows, но их сопротивление оказалось бесполезным. В этой главе предлагается совершить небольшой экскурс в Windows и вкратце рассматриваются следующие вопросы.

- История игр
- Типы игр
- Элементы программирования игр
- Используемые для программирования игр инструменты
- Простая игра FreakOut

Небольшая история

Все началось где-то в 1960-х годах, во времена первых мэйнфреймов. Мне кажется (но, пожалуйста, не ссылайтесь на мои слова), что первой компьютерной игрой была Core Wars ("Войны в памяти") на машинах под управлением Unix. В 1970-х годах появилось множество текстовых и графических (впрочем, это была очень "сырая" графика) приключенческих (так называемых бродилок) игр для различных типов мэйнфреймов и мини-компьютеров.

Как ни странно, но большинство игр были сетевыми! Мне кажется, что до 90% игр составляли многопользовательские игры типа MUD, имитаторы наподобие Star Trek или военные имитаторы. Однако широкие массы не изведали вкуса игр до тех пор, пока не появилась игра Нолана Башнелла (Nolan Bushnell) Pong, которая и открыла эру видеоигр.

Тогда, в 1976–78-х годах, TRS-80, Apple и Atari 800 представляли весь рынок компьютеров. Каждый из них имел свои за и против. Atari 800 был самым мощным (я уверен, что мог бы написать версию Wolfenstein для этого компьютера), TRS-80 в наибольшей степени подходил для бизнеса, а Apple был самым "раскрученным".

Понемногу игры завоевывали рынок программного обеспечения для этих систем, и все большее число подростков играли в них ночи напролет. В те дни компьютерные игры выглядели именно как компьютерные игры, и лишь немногие программисты знали, как они создаются. Книг на эту тему не было вовсе. Все, что было, — это самиздатовские брошюры, содержавшие некоторые части огромной мозаики, да редкие статьи в журнале *Byte*.

Восьмидесятые годы ознаменовались выходом на сцену 16-битовых компьютеров типа IBM PC (и совместимых с ним), Mac, Atari ST, Amiga 500 и т.п. Пришло время, когда игры стали выглядеть хорошо. Появились даже первые трехмерные (3D) игры типа Wing Commander и Flight Simulator, но PC в то время нельзя было назвать игровым компьютером. В 1985 году эту нишу прочно занимали Amiga 500 и Atari ST. Но PC постепенно набирал популярность благодаря дешевизне и применимости в бизнесе. Вывод из всего этого: в конечном счете всегда побеждает компьютер, захвативший рынок, независимо от его технологичности или качества.

В начале 90-х годов несомненным лидером на рынке стал IBM PC-совместимый компьютер. С выходом Microsoft Windows 3.1 он окончательно превзошел Apple Macintosh, став "компьютером делового человека". И все же PC отставал от своих конкурентов в части графики и аудио, что никак не давало играм для него выглядеть так же хорошо, как играм на Amiga.

И вот — да будет свет!..[1]

В конце 1993 года Id Software выпустила продолжение игры Wolfenstein 3D (одна из первых 3D-игр, также разработанная Id Software) — DOOM. PC стал компьютером, на котором можно не только работать, но и играть, компьютером для работы и для дома. DOOM доказала (если это было все еще кому-то не ясно), что PC можно заставить делать все. Это очень важный момент. Запомните его. Нет никакой замены воображению и решимости. Если вы верите в то, что нечто возможно, — это так и есть!

После невероятного успеха DOOM Microsoft начала пересмотр своей позиции относительно игр и их программирования. Стало очевидно, что индустрия развлечений огромна и будет только расти, и Microsoft решила занять достойное место в этой индустрии.

Проблема была в том, что даже Windows 95 имела очень слабые видео- и аудиовозможности, так что Microsoft разработала специальное программное обеспечение Win-G для решения вопросов, связанных с видео. Win-G была заявлена как основная подсистема программирования игр и графики. По сути, она представляла собой всего лишь набор графических вызовов для вывода растровых изображений, и примерно через год Microsoft буквально отрицала ее существование — поверьте, это чистая правда!

Однако к тому времени уже вовсю шла работа над новой системой, которая должна была объединять графику, звук, ввод информации, сетевые возможности и 3D-системы. Так родился продукт DirectX. Как обычно, маркетологи Microsoft немедленно заявили, что DirectX и есть окончательное решение всех проблем программирования игр на платформе PC и что игры в Windows будут такими же быстрыми (и даже быстрее), чем игры под DOS32. Но этого не произошло.

Первые версии DirectX потерпели фиаско, но как программный продукт, а не как технология. Microsoft просто недооценила сложность программирования видеоигр. Но уже с DirectX 3.0 этот продукт начинает работать лучше, чем DOS! Тем не менее большинство компаний, разрабатывающих игры, в те годы (1996–1997) продолжали работать с DOS32 и не спешили переходить к DirectX, пока не вышла версия 5.0 этого продукта.

Используя технологию DirectX, вы можете создать виртуальную DOS-подобную машину с адресным пространством в 4 Гбайт (или больше) и линейной памятью и программировать так, как вы бы это делали, работая под DOS (если, конечно, это то, что вам нравится).

---

[1] Автор цитирует строчку из известной эпиграммы: "*Был этот мир глубокой тьмой окутан. Да будет свет! И вот явился Ньютон*". — *Прим. перев.*

Но что гораздо важнее, теперь у вас есть возможность немедленно работать с вновь появляющимися технологиями графики и звука благодаря особенностям разработки DirectX. Впрочем, об этом мы еще поговорим в нашей книге, а пока вернемся к истории.

Первой игрой нового поколения была DOOM, использовавшая программную растеризацию. Взгляните на рис. 1.1, где приведена копия экрана игры Rex Blade, клона DOOM. Следующее поколение 3D-игр, например Quake I, Quake II или Unreal, по сути, стало "квантовым скачком" в технологии. Взгляните на рис. 1.2, где показана копия экрана игры Unreal. Эта игра и подобные ей используют программную растеризацию совместно с аппаратным ускорением для получения лучшего из этих двух миров. Смею вас заверить, Unreal II или Quake III, работающие на Pentium IV 2.4 GHz с видеокартой GeForce IV TI, выглядят очень привлекательно.

А что же остается нам? Конечно, игры типа Quake и Unreal требуют годы работы больших коллективов, но я надеюсь, что вы, по крайней мере, сможете принять участие в подобных проектах.

На этом с историческими уроками покончено. Поговорим о более близких вещах.

Рис. 1.1. Rex Blade: первое поколение технологии DOOM

Проектирование игр

Один из самых сложных этапов написания видеоигр — их проектирование. Конечно, трехмерная математика сложна, но не менее важно подумать о привлекательности игры, ее сложности, используемом языке (включая сленг и жаргоны). Иначе кто оценит так тщательно разработанные вами объемные следы плазменных зарядов?

Типы игр

В настоящее время существует множество игр, которые можно условно разделить на несколько типов.

DOOM-подобные игры. В большинстве своем это полностью трехмерные игры, типичными примерами которых являются DOOM, Hexen, Quake, Unreal, Duke Nukem 3D и Dark Forces. Технологически это наиболее сложные в разработке игры, требующие хорошего знания различных технологий.

Спортивные игры. Могут быть как двухмерными, так и трехмерными, но все чаще и чаще они становятся трехмерными. В любом случае спортивные игры бывают как для команды, так и для одного человека. Графика в спортивных играх прошла долгий путь, и хотя она не столь впечатляюща, как в DOOM-подобных играх, от этого она не становит-

ся менее захватывающей. Однако наиболее сильной стороной спортивных игр является система искусственного интеллекта, который в играх этого класса обычно наиболее развит по сравнению с другими играми.

Рис. 1.2. Копия экрана игры Unreal

Боевые игры. Обычно предназначены для одного или двух игроков, и их действие рассматривается со стороны либо при помощи подвижной 3D-камеры. Изображение может быть двухмерным, 2.5D (множественные растровые 2D-изображения трехмерных объектов) или полностью трехмерным. Игры этого типа не так популярны на платформе PC, вероятно, из-за проблем интерфейса с контроллерами и в связи с предназначением игр для двух игроков.

Лабиринты. Типичные представители этой старой школы игр — Asteroids, Pac Man, Jazz Jackrabbit. Обычно это двухмерные игры, хотя в последнее время появились их трехмерные версии; правда, сценарий таких игр тот же, что и у двухмерных.

Механические имитаторы. Это игры, осуществляющие некоторый вид вождения, полета, плавания и подобных действий. В большинстве случаев это трехмерные игры, и таковыми они были всегда (хотя и выглядели до последнего времени существенно хуже, чем сейчас).

Имитаторы экосистем. Это новый класс игр, не имеющих аналогов в реальном мире (если не рассматривать таковым весь мир). В качестве примера можно привести Populous, SimCity, SimAnt, Civilization и др. Эти игры позволят вам убедиться, трудно ли быть Богом и управлять некой искусственной системой — городом, колонией или целым миром. Сюда же входят игры — имитаторы экономической системы (например, Gazzillonaire; кстати, очень неплохая игра).

Стратегические или военные игры. Подразделяются на ряд подтипов, перечислять которые здесь не имеет смысла. Сюда входят такие игры, как Warcraft, Diablo, Final Fantasy и т.п. Некоторые из этих игр являются играми реального времени, но при этом требуют наличия стратегического мышления; другие — типичные представители игр, в которых действие осуществляется на основе ходов противников.

Интерактивные повести. Обычно эти игры содержат предварительно созданную графику и видео, и вы путешествуете по заранее созданным дорогам и рассматриваете заранее созданные виды, разгадывая различные головоломки. Обычно такие игры не дают особой свободы выбора перемещений.

Ретро-игры. Это игры для тех, кто помнит старые добрые времена и предпочитает игры тех лет. Зачастую это переделки старых игр на новый лад, т.е. игры с тем же содержанием, но с использованием современных возможностей.

Головоломки и настольные игры. Здесь может быть все, что угодно: двухмерность, трехмерность, предварительный рендеринг и т.п. Tetris, Monopoly, Mahjong — вот некоторые из огромного списка игр, входящих в эту категорию.

Мозговой штурм

После того как вы решите, игру какого типа хотите создать (что обычно просто, так как все мы знаем, что нам нравится), наступает время глубоких размышлений над будущей игрой. Это тот момент, когда вы остаетесь один на один с самим собой и дать вам совет просто некому. К сожалению, не существует способа постоянной генерации идей хороших игр.

Вы должны придумать игру, которая понравится не только вам, но и другим людям и которую можно реально создать. Конечно, вполне можно использовать в качестве точки отсчета другие игры, но избегайте точного копирования чужого продукта. Почитайте научную фантастику и игровые журналы, выясните, какие игры имеют наиболее высокий уровень продаж. Посмотрите различные фантастические кинофильмы — возможно, кроме идей по поводу игры вы сможете почерпнуть и идеи по поводу красивых визуальных эффектов.

Лично я обычно сижу с друзьями (или даже один) и выдаю множество идей, которые представляются мне привлекательными. Затем я развиваю эти идеи до тех пор, пока они не становятся достаточно правдоподобными или пока не доходят до абсурда и не разваливаются на глазах. Порой очень обидно быть переполненным идеями и через пару часов оказаться с пустыми руками. Не отчаивайтесь — так и должно быть. Если идея, пережив ночь, на следующее утро все еще кажется вам привлекательной — у нее есть шансы быть воплощенной в жизнь.

ВНИМАНИЕ

Читайте внимательно, так как я хочу сказать нечто важное. Не откусывайте больше, чем сможете пережевать! Я получаю тысячи писем от новичков, которые в качестве своей первой игры твердо намерены создать игру уровня DOOM или Quake. Этого просто не может произойти. Вы должны быть счастливы, если вам удастся создать клон Asteroids за три-шесть месяцев работы, так что старайтесь не переоценить себя и ставьте перед собой реальные цели. Думайте о том, что в состоянии сделать лично вы, поскольку в конце работы, скорее всего, вы останетесь в одиночестве — все ваши помощники просто улизнут от нее. Кроме того, идея вашей первой игры должна быть простой.

Теперь перейдем к более детальному рассмотрению вопросов.

Разработка документации и сценария

После того как у вас появилась идея, вы должны изложить ее на бумаге. Сейчас, когда я занимаюсь большими игровыми продуктами, я просто вынужден создавать настоящую проектную документацию, но для небольших игр сойдет и несколько страниц детального описания. В основном документация представляет собой карту или набросок игры. В ней должно быть перечислено как можно больше детальной информации об игре, ее уровнях, правилах и т.д. Тогда вы точно знаете, чего хотите, и можете следовать изложенному на бумаге плану. В противном случае детали будут все время изменяться, и в конечном счете ваша игра окажется непоследовательной и бессвязной.

Обычно для начала я записываю небольшую историю: примерно одну-две страницы описания, о чем же эта игра, кто ее основной персонаж, в чем состоит ее идея. И, наконец, как выиграть в этой игре. Затем я принимаю решение об уровнях игры и ее правилах и переношу их на бумагу как можно более детально. После того как это сделано, я могу добавлять или уда-

лять из этого описания те или иные детали, но у меня все равно остается рабочий план. Даже если я придумаю сто новых идей, я не забуду о них и смогу всегда добавить их в игру.

Не пренебрегайте набросками — они здорово помогут вам в работе. На рис. 1.3 показан пример простейших набросков сценария игры. Как видите, ничего сложного: просто смотри на рисунки и работай над ними.

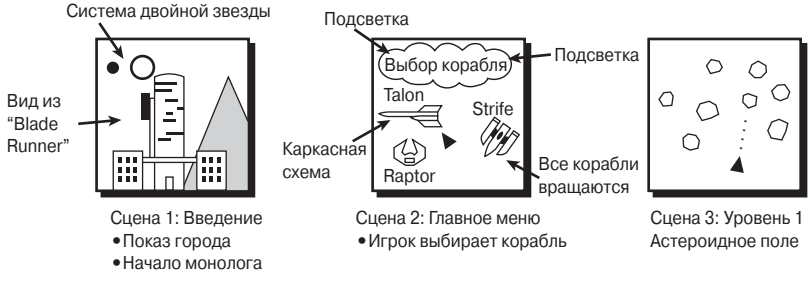


Рис. 1.3. Наброски сценария игры

Жизнеспособность игры

Последняя часть разработки игры — проверка ее жизнеспособности. Вы в самом деле уверены, что ваша игра понравится людям? Не лжете ли вы самому себе? Это весьма серьезный вопрос. Имеется около 10000 различных игр и около 9900 компаний, занимающихся ими, — о такой конкуренции стоит задуматься заранее. Думайте не о том, нравится ли эта игра вам, а о том, как заставить ее понравиться другим.

Ничто, кроме глубоких раздумий и усиленного бета-тестирования игры, не сможет в данном случае вам помочь. Подумайте о привлекательных деталях, на которые обычно ловится много игроков, — вплоть до отделки кабины пилота мореным дубом.

Компоненты игры

Теперь пришло время узнать, чем же видеоигра отличается от программ других типов. Видеоигры — исключительно сложная разновидность программного обеспечения и, вне всякого сомнения, наиболее трудная для написания. Конечно, написать MS Word сложнее, чем Asteroids, но я пока не встречал программы, которая была бы сложнее Unreal.

Это означает, что вы должны изучить новый способ программирования, в большей степени подходящий для приложений реального времени и имитаторов, чем построчная, управляемая событиями или последовательная логика программ, над которыми вы работали ранее. Видеоигра представляет собой в основном непрерывный цикл, который осуществляет работу логики программы и вывод на экран, обычно со скоростью 30 кадров в секунду (frames per second, fps) или выше. Скорость вывода близка к скорости показа кинофильма, с той разницей, что в данном случае кинофильм создается по мере его показа.

Взглянем на упрощенный цикл игры, показанный на рис. 1.4. Далее описаны все его части.

Инициализация

В этой части вы выполняете стандартные операции, обычные для любой программы, такие, как распределение памяти, захват ресурсов, загрузка данных с диска и т.п.

Вход в цикл игры

В этой части выполнение кода входит в основной цикл игры. Здесь начинается собственно действие игры, которое продолжается до тех пор, пока пользователь не выйдет из основного цикла.

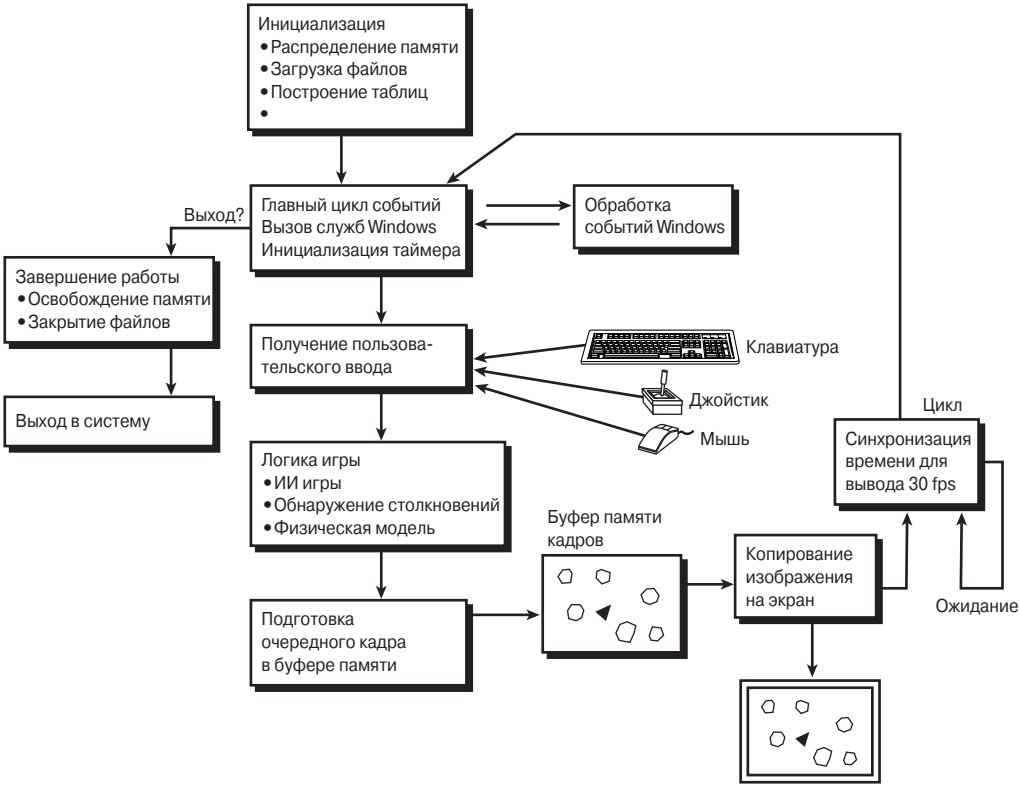


Рис. 1.4. Обобщенная архитектура цикла игры

Получение пользовательского ввода

В этой части ввод пользователя обрабатывается и/или буферизуется для последующего использования системой искусственного интеллекта (ИИ) и логики игры.

ИИ и логика игры

Эта часть содержит основной код игры. Именно здесь выполняется работа искусственного интеллекта, логики игры и физической модели, результаты которой используются для вывода на экран очередного кадра.

Подготовка очередного кадра

В этой части результаты ввода пользователя и работы искусственного интеллекта, логики игры и физической модели используются для подготовки очередного кадра анимации игры. Обычно это изображение подготавливается в отдельном буфере памяти, не связанном с экраном, так что процесс подготовки пользователю не виден. Затем подготовленное изображение быстро выводится на экран.

Синхронизация вывода

Скорость работы компьютера зависит от уровня сложности игры. Например, если на экране происходит действие, включающее 1000 объектов, процессор явно будет загружен существенно сильнее, чем при наличии всего 10 объектов. Соответственно варьируется

частота вывода кадров, что недопустимо. Следовательно, требуется некий механизм синхронизации, обеспечивающий постоянную частоту кадров (обычно 30 fps считается оптимальной скоростью вывода).

Цикл

Эта часть кода предельно проста — она обеспечивает возврат к началу цикла и выполнение очередного прохода.

Завершение работы

Эта часть кода означает завершение работы игры и возврат в операционную систему. Однако до возврата вы должны освободить все захваченные ресурсы, как и при возврате из любой другой программы.

Конечно, приведенная схема несколько упрощена, но указывает все существенные части игры. В большинстве случаев игра представляет собой конечный автомат, имеющий ряд состояний. В листинге 1.1 цикл показан более детально, примерно так, как он мог бы выглядеть в реальной игре.

Листинг 1.1. Простой цикл событий игры

```
// Определения состояний цикла игры
#define  GAME_INIT          // Инициализация игры
#define  GAME_MENU          // Игра в режиме меню
#define  GAME_STARTING      // Игра начинает работу
#define  GAME_RUN           // Игра работает
#define  GAME_RESTART       // Перезапуск игры
#define  GAME_EXIT          // Выход из игры

// Глобальные переменные игры
int game_state = GAME_INIT;  // Начальное состояние игры
int error      = 0;          // Возврат ошибки ОС

void main()
{
// Реализация основного цикла игры

  while( game_state != GAME_EXIT )
  {
    // Какое текущее состояние игры?
    switch( game_state )
    {
      case GAME_INIT: // Инициализация игры
        {
          // Распределение памяти
          // и захват ресурсов
          Init();

          // переход в состояние меню
          game_state = GAME_MENU;
        } break;

      case GAME_MENU: // Игра в режиме меню
        {
          // Вызов функции меню
```

```
      // и переключение состояния
      game_state = Menu();
    } break;

  case GAME_STARTING: // Игра начинает работу
    {
      // Это необязаьельное состояние, которое
      // обычно используется для
      // дополнительной работы с захваченными
      // ресурсами
      Switch_For_Run();

      // Переход в состояние работы
      game_state = GAME_RUN;
    } break;

  case GAME_RUN: // Игра работает
    {
      // Эта часть содержит логику игры

      // Очистка экрана
      Clear();

      // Получение ввода
      Get_Input();

      // Работа логики и ИИ
      Do_Logic();

      // Вывод очередного кадра
      Render_Frame();

      // Синхронизация
      Wait();

      // Единственный способ изменения этого
      // состояния - через взаимодействие с
      // пользователем (либо, возможно,
      // вследствие проигрыша)
    } break;

  case GAME_EXIT: // Выход из игры
    {
      // Раз мы здесь, работа программы
      // успешно завершается и мы должны
      // освободить ресурсы и выполнить
      // остальную работу этого типа
      Release_And_Cleanup();

      // Раз мы здесь, ошибок не было
      error = 0;

      // Здесь переключение состояний не
      // требуется, так как все равно
      // осуществляется выход из программы
    } break;
```

```
        default: break;
      } // switch
   }    // while

      // Возврат кода ошибки операционной системе
   return( error );
}    // main
```

Хотя листинг 1.1 и нефункционален, я думаю, что основную идею вы уловили. Все циклы игр в той или иной степени следуют приведенной схеме. Взгляните также на рис. 1.5, где приведена диаграмма переходов конечного автомата игры. Как видите, все более-менее просто.

Позже в этой главе при рассмотрении демонстрационной игры FreakOut мы еще поговорим о циклах и конечных автоматах.

Общие рекомендации по программированию игр

Теперь я хочу поговорить о некоторых общих принципах и методах программирования, о которых не стоит забывать и которым необходимо по возможности следовать для того, чтобы программирование игр стало более простой задачей.

Начнем с того, что видеоигры представляют собой программы с исключительно высокой производительностью. В наиболее критических по времени работы или требованиям к памяти частях кода больше нельзя использовать API высокого уровня. Все, что связано с внутренним циклом вашей игры, должно быть написано вами самостоятельно, иначе неизбежны проблемы с производительностью игры. Это, понятно, не означает, что вы не должны использовать API DirectX, поскольку DirectX разрабатывался с учетом требования максимальной производительности. Но, вообще говоря, избегайте вызовов функций высокого уровня.

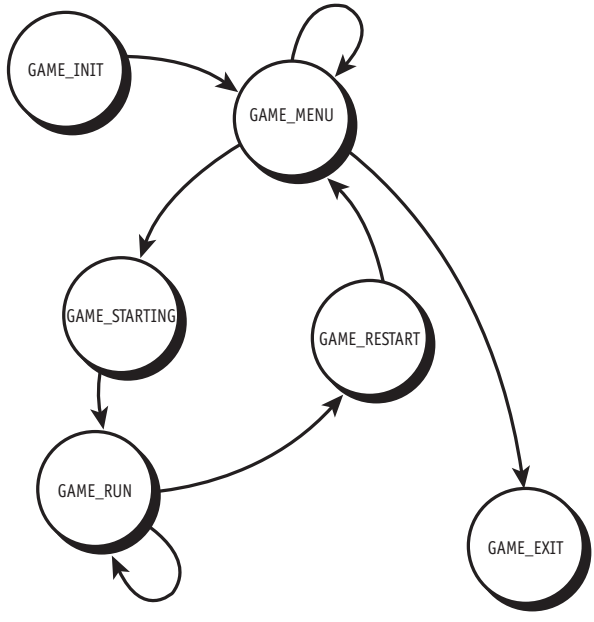


Рис. 1.5. Диаграмма переходов в цикле игры

Помня об этом, рассмотрим ряд приемов, которые не стоит забывать при работе над играми.

СЕКРЕТ

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

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

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

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

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

СЕКРЕТ

Используйте встроенные функции. Предыдущий фрагмент можно еще больше улучшить с помощью директивы inline, которая полностью устраняет код вызова функции, указывая компилятору на необходимость размещения кода функции непосредственно в месте ее вызова. Конечно, это несколько увеличивает программу, но скорость для нас гораздо важнее[2].

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

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

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

СЕКРЕТ

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

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

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

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

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

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

C++

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

СЕКРЕТ

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

СЕКРЕТ

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

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

используйте код попроще:

```
x += 2*buffer[index];
index++;

if (x > 10)
{
```

```
  // Выполняем некоторые действия
} // if
```

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

СЕКРЕТ

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

```
int y_pos = 10;

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


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

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

СЕКРЕТ

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

СЕКРЕТ

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

СЕКРЕТ

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

СЕКРЕТ

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

СЕКРЕТ

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

СЕКРЕТ

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

СЕКРЕТ

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

Использование инструментов

В прошлом создание игр не требовало иного инструментария, кроме простого текстового редактора и, возможно, простого самодельного графического редактора. Но сегодня создание игр — задача более сложная. Как минимум, вам нужен компилятор C/C++, графический редактор и программа для обработки звука. Кроме того, вам может потребоваться программа для моделирования трехмерных объектов, а также программа для работы с музыкой, если вы используете в своей игре возможности MIDI.

Компиляторы C/C++

Для разработки программного обеспечения для Windows 95/NT наилучший компилятор — MS VC++ 6.0+. Он может выполнить все, что вам потребуется, и даже больше. Генерируемые им EXE-файлы содержат наиболее быстрый код. Компилятор Borland также неплох (и существенно дешевле), но его возможности гораздо ниже. В любом случае полные версии ни того и ни другого вам не нужны. Студенческой версии, способной создавать EXE-файлы Win32, более чем достаточно.

Программное обеспечение для двухмерной графики

При работе над играми вам нужны программы для рисования, черчения и обработки изображений. Программы для рисования позволяют пиксель за пикселем создавать изображение из графических примитивов и управлять им. С точки зрения отношения цена/производительность лидером мне представляется Paint Shop Pro. Неплох ProCreate Painter, но он предназначен для художников-профессионалов, не говоря уже о его цене. Я лично предпочитаю Corel Photo-Paint, но его возможности определенно выше требований новичков.

Программы для черчения также позволяют создавать изображения, но из кривых, линий и двухмерных геометрических примитивов. Эти программы не так необходимы, но если вам потребуется программа такого типа, рекомендую воспользоваться Adobe Illustrator.

Последними в списке программ для работы с двухмерным изображением упоминаются программы для обработки изображений, среди которых я назвал бы Adobe Photoshop и Corel Photo-Paint. Хотя лично мне больше нравится последняя, выбор за вами.

Программное обеспечение для обработки звука

Девяносто процентов звуковых эффектов (SFX), используемых сегодня в играх, представляют собой цифровые образцы звука. Для работы с ними вам потребуется программа обработки звука. Наилучшая программа для этого — Sound Forge Xp. При наличии множества возможностей по обработке звука эта программа остается очень простой в использовании.

Моделирование трехмерных объектов

Обычно такого рода программное обеспечение стоит слишком дорого, но в последнее время мне попались несколько относительно недорогих программ моделирования трехмерных объектов, достаточно мощных, чтобы создавать видеоклипы. Лично я пользуюсь Caligari TrueSpace, которая стоит всего лишь несколько сотен долларов.

Если вы хотите большей мощности и натурализма — обратитесь к 3D Studio Max, которая, правда, стоит около $2500. Однако мы в основном будем использовать данное

программное обеспечение для создания трехмерных каркасов, а не реалистичных изображений, так что Caligari TrueSpace будет вполне достаточно.

Музыкальные и MIDI-программы

На сегодня в играх используются два вида музыки: чисто цифровая (типа CD) и MIDI (musical instrument digital interface — цифровой интерфейс музыкальных инструментов), представляющая собой синтезированный на основе нотного представления звук. Для работы с MIDI вам требуется соответствующее программное обеспечение, и здесь я бы посоветовал воспользоваться программой Cakewalk. Более подробно о MIDI речь идет в главе 10, "DirectSound и DirectMusic".

Использование компилятора

Зачастую в изучении программирования игр для Windows крайне сложно научиться правильно пользоваться компилятором. Очень часто вы бодро запускаете среду программирования, запускаете компиляцию, и ваш экран переполняется сообщениями об ошибках. Чтобы помочь вам, рассмотрим некоторые основные концепции.

1. Пожалуйста, прочитайте документацию к компилятору, ну пожалуйста, пожалуйста! Я вас очень прошу!
2. Вы должны установить на машине DirectX SDK. Для этого требуется лишь перейти в каталог `DirectX` на компакт-диске, прочитать файл `README.TXT` и выполнить все, что там сказано (а именно запустить программу`INSTALL.EXE`).
3. Мы будем создавать `EXE`-программы Win32, а не DLL, компоненты ActiveX и т.п. Поэтому первое, что вам нужно, — создать новый проект и установить тип создаваемого файла Win32 Application. Этот шаг при работе с компилятором VC++ 6.0 показан на рис. 1.6.
4. Добавьте в проект исходные файлы при помощи пункта Add Files из главного меню. Это действие показано на рис. 1.7.
5. Приступив к работе с DirectX, мы должны включить в проект большинство из перечисленных далее и представленных на рис. 1.8 библиотек COM интерфейсов DirectX.
   - `DDRAW.LIB`
   - `DSOUND.LIB`
   - `DINPUT.LIB`
   - `DINPUT8.LIB`
   - `DSETUP.LIB`

НА ЗАМЕТКУ Если вы не используете DirectSetup, библиотека `DSETUP.LIB` вам не потребуется.

   Эти библиотечные файлы DirectX после установки DirectX SDK находятся в каталоге `LIB`. Вы *должны* явно добавить эти файлы в ваш проект; добавления пути к файлам недостаточно, так как компилятор при этом будет, скорее всего, находить старые файлы от DirectX 3.0 из поставки компилятора. Кроме того, вам следует добавить в проект библиотеку расширений Windows мультимедиа `WINMM.LIB`, которая находится в каталоге `LIB` вашего компилятора.
6. Вы готовы к компиляции программы.

ВНИМАНИЕ

Если вы используете Borland, то в поставке DirectX SDK для вас предназначен специальный каталог с этим именем, и добавлять в свой проект библиотечные файлы вы должны именно из этого каталога.

Если после прочтения этого раздела у вас остались вопросы — ничего страшного. Мы еще вернемся к вопросам использования компилятора при рассмотрении программирования в Windows и изучении DirectX.

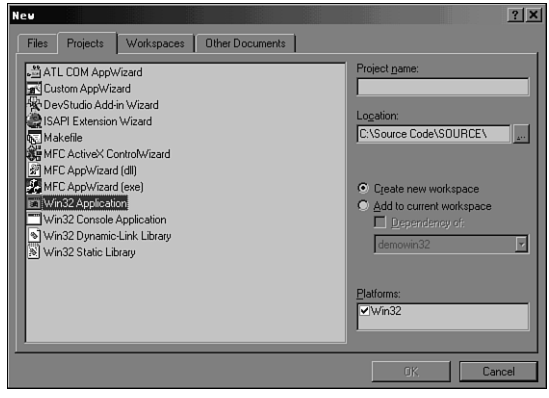


Рис. 1.6. Создание Win32 EXE-файла в Visual C++ 6.0

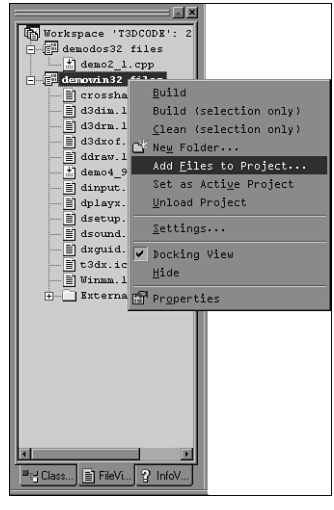


Рис. 1.7. Добавление файлов в проект Visual C++ 6.0

Пример: игра FreakOut

Пока мы еще окончательно не свихнулись со всеми этими отвлеченными разговорами о Windows, DirectX и 3D-графике, давайте сделаем остановку и рассмотрим полностью готовую игру — конечно, простенькую, но несомненно игру. Здесь вы увидите цикл игры, некоторые графические вызовы и то, как компилируется игра.

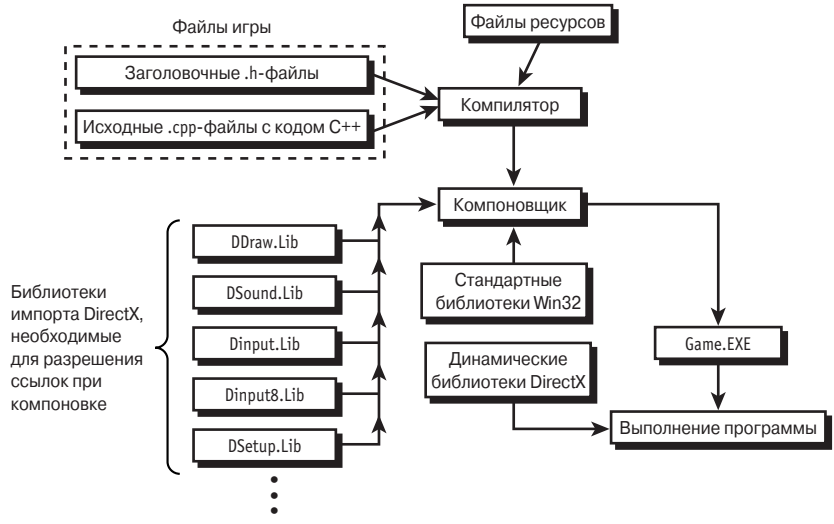


Рис. 1.8. Ресурсы, необходимые для создания приложения Win32 DirectX

Проблема только в том, что мы все еще находимся в главе 1, а в игре используются материалы из последующих глав. Давайте договоримся так: в игре мы просто используем API *черного ящика*. Исходя из этой посылки, я задамся вопросом: "Каков абсолютный минимум требований при создании двухмерной игры типа Breakout?" Все, что нам действительно нужно, — это следующая функциональность:

- переход в произвольный графический режим;
- вывод цветных прямоугольников на экран;
- получение клавиатурного ввода;
- синхронизация цикла игры с использованием некоторых функций работы с таймером;
- вывод цветной строки текста на экран.

В связи с этим я создал библиотеку BLACKBOX.CPP|H, которая содержит набор функций DirectX (точнее, только функций DirectDraw) вместе с кодом, реализующим требующуюся нам функциональность. Поэтому пока что вам не нужно разбираться с тем, как именно работают эти функции, прототипы которых имеются в файле BLACKBOX.H; вы должны просто использовать их и не забыть добавить в проект файлы BLACKBOX.CPP|H.

Используя библиотеку BLACKBOX, я написал игру FreakOut, которая демонстрирует ряд концепций, обсуждаемых в этой главе. Во FreakOut имеются все основные компоненты реальных игр, включая цикл игры, подсчет очков, систему уровней и даже физическую модель мяча (впрочем, чрезвычайно примитивную). На рис. 1.9 показана копия экрана игры. Конечно, это не Arkanoid, но для четырех часов работы, согласитесь, это не так уж и плохо.

Прежде чем перейти к исходному тексту игры, взгляните на проект игры и различные его компоненты (рис. 1.10).

Как видно из рисунка, игра состоит из следующих файлов.

FREAKOUT.CPP	Основная логика игры, использующая BLACKBOX.CPP и создающая минимальное приложение Win32
BLACKBOX.CPP	Библиотека игры (пока мы ее не рассматриваем)
BLACKBOX.H	Заголовочный файл библиотеки BLACKBOX

DDRAW.LIB — Библиотека импорта DirectDraw, необходимая для компоновки приложения. Эта библиотека не содержит реальный код DirectX, а представляет собой необходимого для компоновки программы посредника, который загружает динамическую библиотеку DDRAW.DLL, выполняющую всю необходимую работу

DDRAW.DLL — Динамическая библиотека DirectDraw, содержащая COM-реализацию функций интерфейса DirectDraw, вызываемых с участием библиотеки импорта DDRAW.LIB. Вам не нужно самим беспокоиться о загрузке этой библиотеки и подобных вещах — просто убедитесь, что в системе установлены файлы времени исполнения DirectX

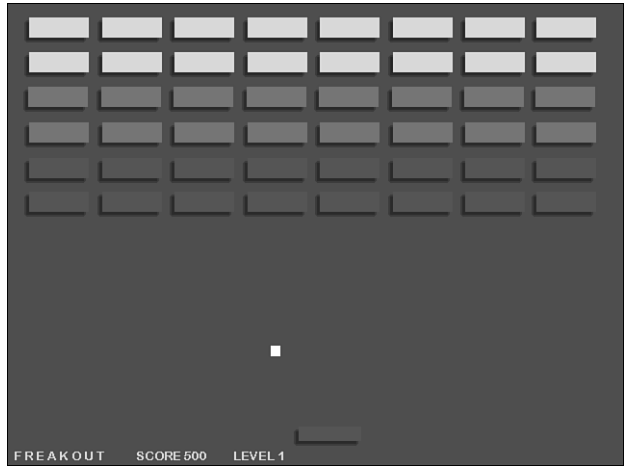


Рис. 1.9. Экран игры FreakOut

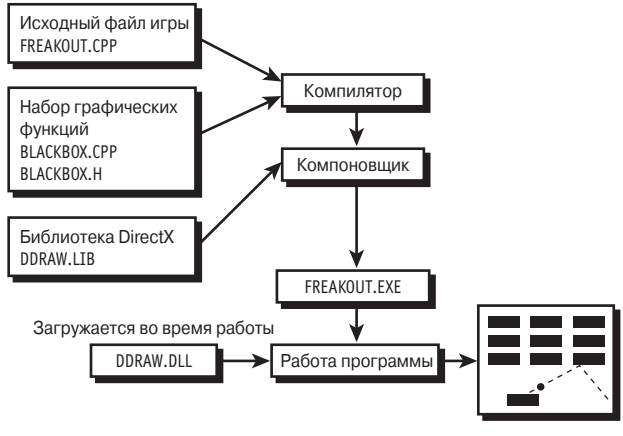


Рис. 1.10. Структура FreakOut

Таким образом, для компиляции необходимо включить в проект исходные файлы BLACKBOX.CPP и FREAKOUT.CPP, скомпоновать его с библиотекой DDRAW.LIB и не забыть убедиться, что BLACKBOX.H находится в рабочем каталоге, чтобы компилятор мог найти его.

Теперь взглянем на файл BLACKBOX.H и выясним, прототипы каких функций в нем имеются (листинг 1.2).

Листинг 1.2. Заголовочный файл BLACKBOX.H

```
// BLACKBOX.H

// Предотвращение многократного включения
#ifndef BLACKBOX
#define BLACKBOX

// ОПРЕДЕЛЕНИЯ ////////////////////////////////////////

// Размер экрана по умолчанию
#define SCREEN_WIDTH   640
#define SCREEN_HEIGHT  480
#define SCREEN_BPP     8  // бит на пиксель
#define MAX_COLORS     256 // Максимальное количество цветов

// MACROS /////////////////////////////////////////////

// Асинхронное чтение клавиатуры
#define KEY_DOWN(vk_code) \
   ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)   \
   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// Инициализация структур DirectDraw
#define DD_INIT_STRUCT(ddstruct) \
   { memset(&ddstruct,0,sizeof(ddstruct)); \
     ddstruct.dwSize=sizeof(ddstruct); }

// Типы ///////////////////////////////////////////////

// Основные беззнаковые типы
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char  UCHAR;
typedef unsigned char  BYTE;

// Внешние объекты ////////////////////////////////////

extern LPDIRECTDRAW7       lpdd;
extern LPDIRECTDRAWSURFACE7  lpddsprimary;
extern LPDIRECTDRAWSURFACE7  lpddsback;
extern LPDIRECTDRAWPALETTE   lpddpal;
extern LPDIRECTDRAWCLIPPER   lpddclipper;
extern PALETTEENTRY        palette[256];
extern PALETTEENTRY        save_palette[256];
extern DDSURFACEDESC2      ddsd;
extern DDBLTFX             ddbltfx;
extern DDSCAPS2            ddscaps;
extern HRESULT             ddrval;
extern DWORD               start_clock_count;

// Прямоугольник обрезки
```

```
extern int min_clip_x,
       max_clip_x,
       min_clip_y,
       max_clip_y;

// Изменяются при вызове DD_Init()
extern int screen_width,    // Ширина экрана
       screen_height,       // Высота экрана
       screen_bpp;          // Бит на пиксель

// Прототипы ////////////////////////////////////////////

// Функции DirectDraw
int DD_Init(int width, int height, int bpp);
int DD_Shutdown(void);
LPDIRECTDRAWCLIPPER DD_Attach_Clipper(
     LPDIRECTDRAWSURFACE7 lpdds,
     int num_rects, LPRECT clip_list);
int DD_Flip(void);
int DD_Fill_Surface(LPDIRECTDRAWSURFACE7 lpdds,int color);

// Функции работы со временем
DWORD Start_Clock(void);
DWORD Get_Clock(void);
DWORD Wait_Clock(DWORD count);

// Графические функции
int Draw_Rectangle(int x1, int y1, int x2, int y2, int color,LPDIRECTDRAWSURFACE7 lpdds=lpddsback);

// Функции gdi
int Draw_Text_GDI(char *text, int x,int y,COLORREF color,
           LPDIRECTDRAWSURFACE7 lpdds=lpddsback);
int Draw_Text_GDI(char *text, int x,int y,int color,
           LPDIRECTDRAWSURFACE7 lpdds=lpddsback);

#endif
```

Пока что не напрягайте излишне свои мозги, пытаясь разобраться со всем этим кодом и глобальными переменными. Просто посмотрите на объявленные в заголовочном файле функции. Как видите, они обеспечивают всю необходимую функциональность для нашего простого графического интерфейса. Теперь с помощью этих функций я создаю игру FREAKOUT.CPP, приведенную в листинге 1.3. Рассмотрите приведенный код, в особенности основной цикл игры и вызовы функций.

Листинг 1.3. Исходный файл FREAKOUT.CPP

```
// FREAKOUT.CPP - демонстрационная игра

// Включаемые файлы /////////////////////////////////////

#define WIN32_LEAN_AND_MEAN      // Включение всех макросов
#define INITGUID                 // Включение графического
                                 // интерфейса пользователя
```

```
#include <windows.h>       // Включение элементов Windows
#include <windowsx.h>
#include <mmsystem.h>

#include <iostream.h>      // Включение функций C/C++
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>         // Включение DirectX
#include "blackbox.h"      // Включение библиотеки

// Определения ////////////////////////////////////////

// Определения окна игры
#define WINDOW_CLASS_NAME "WIN3DCLASS"  // Имя класса

#define WINDOW_WIDTH        640              // Размер окна
#define WINDOW_HEIGHT       480

// Состояния цикла игры
#define GAME_STATE_INIT     0
#define GAME_STATE_START_LEVEL 1
#define GAME_STATE_RUN      2
#define GAME_STATE_SHUTDOWN   3
#define GAME_STATE_EXIT     4

// Определения блоков
#define NUM_BLOCK_ROWS      6
#define NUM_BLOCK_COLUMNS    8

#define BLOCK_WIDTH        64
#define BLOCK_HEIGHT       16
#define BLOCK_ORIGIN_X      8
#define BLOCK_ORIGIN_Y      8
#define BLOCK_X_GAP        80
#define BLOCK_Y_GAP        32

// Определения ракетки
#define PADDLE_START_X      (SCREEN_WIDTH/2 - 16)
#define PADDLE_START_Y      (SCREEN_HEIGHT - 32);
#define PADDLE_WIDTH        32
#define PADDLE_HEIGHT       8
#define PADDLE_COLOR        191
```

```
// Определения мяча
#define BALL_START_Y        (SCREEN_HEIGHT/2)
#define BALL_SIZE           4

// Прототипы ////////////////////////////////////////////

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

// Глобальные переменные ////////////////////////////////

HWND main_window_handle  = NULL; // Дескриптор окна
HINSTANCE main_instance  = NULL;  // Экземпляр
int game_state        = GAME_STATE_INIT;
                      // Начальное состояние

int paddle_x = 0, paddle_y = 0;  // Позиция ракетки
int ball_x   = 0, ball_y   = 0;       // Позиция мяча
int ball_dx  = 0, ball_dy  = 0;     // Скорость мяча
int score    = 0;                      // Счет
int level    = 1;                      // Текущий уровень
int blocks_hit = 0;                    // Количество выбитых блоков

// Сетка игры

UCHAR blocks[NUM_BLOCK_ROWS][NUM_BLOCK_COLUMNS];

// Функции //////////////////////////////////////////////

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

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

   case WM_PAINT:
       {
       // Рисование
       hdc = BeginPaint(hwnd,&ps);
```

```
      // Теперь окно действительно

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

   case WM_DESTROY:
     {
     // Уничтожение приложения
     PostQuitMessage(0);
     return(0);
     } break;

   default:break;

   } // switch

// Обработка по умолчанию остальных сообщений
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // WinProc

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

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

WNDCLASS winclass;    // Класс, создаваемый нами
HWND     hwnd;        // Дескриптор окна
MSG      msg;         // Сообщение
HDC      hdc;         // Контекст устройства
PAINTSTRUCT ps;

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

// Регистрация класса окна
if (!RegisterClass(&winclass))
```

```
  return(0);

// Создание окна (обратите внимание
// на использование WS_POPUP)
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME,     // Класс
        "WIN3D Game Console",                    // Заголовок
        WS_POPUP | WS_VISIBLE,
        0,0,                                     // Координаты
        GetSystemMetrics(SM_CXSCREEN),   // ширина
        GetSystemMetrics(SM_CYSCREEN),   // высота
        NULL,        // Дескриптор родителя
        NULL,        // Дескриптор меню
        hinstance,   // Экземпляр
        NULL)))      // Параметры создания
return(0);

// Сокрытие мыши
ShowCursor(FALSE);

// Сохранение дескриптора окна и экземпляра
main_window_handle = hwnd;
main_instance     = hinstance;

// Инициализация консоли игры
Game_Init();

// Главный цикл игры
while(1)
  {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // Проверка на выход
    if (msg.message == WM_QUIT)
      break;

    // Трансляция клавиш
    TranslateMessage(&msg);

    // Пересылка сообщения процедуре окна
    DispatchMessage(&msg);
    } // if

  // Основная процедура игры
  Game_Main();

  } // while

// Завершение игры и освобождение ресурсов
Game_Shutdown();

// Возврат мыши
ShowCursor(TRUE);
```

```
// Выход в Windows
return(msg.wParam);

} // WinMain

// Консольные функции T3DX ////////////////////////////////

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


// Успешное завершение
return(1);

} // Game_Init

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

int Game_Shutdown(void *parms)
{
// Завершение игры и освобождение ресурсов

// Успешное завершение
return(1);

} // Game_Shutdown

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

void Init_Blocks(void)
{
// Инициализация поля блоков
for (int row=0; row < NUM_BLOCK_ROWS; row++)
   for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
      blocks[row][col] = row*16+col*3+16;

} // Init_Blocks

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

void Draw_Blocks(void)
{
// Вывод блоков на экран
int x1 = BLOCK_ORIGIN_X, // Отслеживание текущего положения
   y1 = BLOCK_ORIGIN_Y;

// draw all the blocks
for (int row=0; row < NUM_BLOCK_ROWS; row++)
   {
   x1 = BLOCK_ORIGIN_X;

   for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
```

```
      {
      if (blocks[row][col]!=0)
        {
        Draw_Rectangle(x1-4,y1+4,
          x1+BLOCK_WIDTH-4,y1+BLOCK_HEIGHT+4,0);

        Draw_Rectangle(x1,y1,x1+BLOCK_WIDTH,
          y1+BLOCK_HEIGHT,blocks[row][col]);
        } // if

      x1+=BLOCK_X_GAP;
      } // for col

    y1+=BLOCK_Y_GAP;

    } // for row

} // Draw_Blocks

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

void Process_Ball(void)
{
// Обработка движения мяча, соударения с ракеткой или
// блоком; отражение мяча и удаление блока с экрана

// Проверка соударения с блоком

// Проверяем все блоки (неэффективно, но просто; позже вы
// познакомитесь с другими способами решения данной задачи)

int x1 = BLOCK_ORIGIN_X,
    y1 = BLOCK_ORIGIN_Y;

int ball_cx = ball_x+(BALL_SIZE/2),  // Вычисляем центр мяча
    ball_cy = ball_y+(BALL_SIZE/2);

// Проверка столкновения с ракеткой
if (ball_y > (SCREEN_HEIGHT/2) && ball_dy > 0)
  {
  int x = ball_x+(BALL_SIZE/2);
  int y = ball_y+(BALL_SIZE/2);

  if ((x >= paddle_x && x <= paddle_x+PADDLE_WIDTH) &&
      (y >= paddle_y && y <= paddle_y+PADDLE_HEIGHT))
      {
      // Отражение мяча
      ball_dy=-ball_dy;
      ball_y+=ball_dy;

      // Изменения траектории из-за движения ракетки
      if (KEY_DOWN(VK_RIGHT))
        ball_dx-=(rand()%3);
```

```
    else
    if (KEY_DOWN(VK_LEFT))
      ball_dx+=(rand()%3);
    else
      ball_dx+=(-1+rand()%3);

    // Проверка, есть ли блоки в системе. Если нет -
    // сообщение для перехода на новый уровень
    if (blocks_hit >= (NUM_BLOCK_ROWS*NUM_BLOCK_COLUMNS))
      {
      game_state = GAME_STATE_START_LEVEL;
      level++;
      } // if

    MessageBeep(MB_OK);

    return;

    } end if

  } end if

// Сканируем все блоки на наличие удара мяча
for (int row=0; row < NUM_BLOCK_ROWS; row++)
   {
   x1 = BLOCK_ORIGIN_X;

   for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
      {
      if (blocks[row][col]!=0)
        {
        if ((ball_cx>x1) && (ball_cx<x1+BLOCK_WIDTH) &&
           (ball_cy>y1) && (ball_cy<y1+BLOCK_HEIGHT))
           {
           // Удаляем блок
           blocks[row][col] = 0;

           // Увеличиваем счетчик удаленных блоков
           blocks_hit++;

           // Изменение траектории мяча
           ball_dy=-ball_dy;

           ball_dx+=(-1+rand()%3);

           MessageBeep(MB_OK);

           // Изменение счета
           score+=5*(level+(abs(ball_dx)));

           return;

           } // if
```

```
        } // if

      x1+=BLOCK_X_GAP;
      } // for col

   y1+=BLOCK_Y_GAP;

   } // for row

} // Process_Ball

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

int Game_Main(void *parms)
{
// Главная функция игры

char buffer[80]; // Используется для вывода текста

// В каком состоянии игра?
if (game_state == GAME_STATE_INIT)
   {
   // Инициализация графики
   DD_Init(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP);

   // Инициализация генератора случайных чисел
   srand(Start_Clock());

   // Позиционирование ракетки
   paddle_x = PADDLE_START_X;
   paddle_y = PADDLE_START_Y;

   // Позиционирование мяча
   ball_x = 8+rand()%(SCREEN_WIDTH-16);
   ball_y = BALL_START_Y;
   ball_dx = -4 + rand()%(8+1);
   ball_dy = 6 + rand()%2;

   // Переход в стартовое состояние
   game_state = GAME_STATE_START_LEVEL;

   } // if
//////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_START_LEVEL)
   {
   // Переход на новый уровень

   // Инициализация блоков
   Init_Blocks();

   // Сброс счетчика блоков
```

```
  blocks_hit = 0;

  // Переход в состояние работы
  game_state = GAME_STATE_RUN;

  } // if
////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_RUN)
  {
  // Запуск часов
  Start_Clock();

  // Очистка экрана
  Draw_Rectangle(0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1,200);

  // Перемещение ракетки
  if (KEY_DOWN(VK_RIGHT))
    {
    paddle_x+=8;

    // Ракетка не должна выходить за экран
    if (paddle_x > (SCREEN_WIDTH-PADDLE_WIDTH))
      paddle_x = SCREEN_WIDTH-PADDLE_WIDTH;

    } // if
  else
  if (KEY_DOWN(VK_LEFT))
    {
    paddle_x-=8;

    // Ракетка не должна выходить за экран
    if (paddle_x < 0)
      paddle_x = 0;

    } // if

  // Рисуем блоки
  Draw_Blocks();

  // Перемещаем мяч
  ball_x+=ball_dx;
  ball_y+=ball_dy;

  // Мяч не должен выходить за пределы экрана
  if (ball_x > (SCREEN_WIDTH - BALL_SIZE) || ball_x < 0)
    {
    ball_dx=-ball_dx;
    ball_x+=ball_dx;
    } // if

  if (ball_y < 0)
    {
```

```
    ball_dy=-ball_dy;
    ball_y+=ball_dy;
    } // if
  else
  // Потеря мяча
  if (ball_y > (SCREEN_HEIGHT - BALL_SIZE))
    {
    ball_dy=-ball_dy;
    ball_y+=ball_dy;
    score-=100;
    } // if

  // Ограничение скорости мяча
  if (ball_dx > 8) ball_dx = 8;
  else
  if (ball_dx < -8) ball_dx = -8;

  // Проверка соударения мяча с ракеткой или блоком
  Process_Ball();

  // Рисуем ракетку
  Draw_Rectangle(paddle_x-8, paddle_y+8,
              paddle_x+PADDLE_WIDTH-8,
              paddle_y+PADDLE_HEIGHT+8,0);

  Draw_Rectangle(paddle_x, paddle_y,
              paddle_x+PADDLE_WIDTH,
              paddle_y+PADDLE_HEIGHT,PADDLE_COLOR);

  // Рисуем мяч
  Draw_Rectangle(ball_x-4, ball_y+4, ball_x+BALL_SIZE-4,
              ball_y+BALL_SIZE+4, 0);
  Draw_Rectangle(ball_x, ball_y, ball_x+BALL_SIZE,
              ball_y+BALL_SIZE, 255);

  // Выводим информацию
  sprintf(buffer,"F R E A K O U T          Score "
              "%d          Level %d",score,level);
  Draw_Text_GDI(buffer, 8,SCREEN_HEIGHT-16, 127);

  // Вывод на экран
  DD_Flip();

  // Синхронизация до 33 fps
  Wait_Clock(30);

  // Нет ли запроса на выход?
  if (KEY_DOWN(VK_ESCAPE))
    {
    PostMessage(main_window_handle, WM_DESTROY,0,0);
    game_state = GAME_STATE_SHUTDOWN;
    } // if
```

```
  } // if
////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_SHUTDOWN)
  {
  // Завершение игры и освобождение ресурсов
  DD_Shutdown();

  // Состояние выхода
  game_state = GAME_STATE_EXIT;

  } // if

// Успешное завершение
return(1);

} // Game_Main

////////////////////////////////////////////////////////
```

Неплохо, правда? Перед вами полный текст игры Win32/DirectX. Ну или почти полный, поскольку есть еще исходный текст библиотеки BLACKBOX.CPP, но будем считать, что это часть DirectX, написанная кем-то посторонним (мною!). А теперь еще раз взгляните на листинг 1.3.

Windows требует, чтобы у нас был *цикл событий*. Это стандартное требование управляемой событиями операционной системы к любой программе Windows. Однако игра не является управляемой событиями — она работает все время, независимо от того, делает ли пользователь что-то в данный момент или нет. Так что нам необходим минимальный цикл событий для удовлетворения требований Windows. Код, который реализует поддержку цикла, находится в WinMain().

Главной входной точкой для всех Windows-программ является WinMain(), подобно тому как main() представляет собой точку входа в программах DOS/Unix. WinMain() создает окно и входит в цикл событий. Если Windows что-то требуется, мы откликнемся на эти требования. Но, когда с обязательной частью работы покончено, мы вызываем Game_Main(). Именно здесь и выполняется основная работа игры.

Если вы хотите, можете использовать внутри Game_Main() цикл и никогда не возвращаться в основной цикл событий в WinMain(). Однако это не лучшая идея, поскольку Windows не сможет получать сообщения и это приведет к "голоданию" системы. Так что мы просто вынуждены после вывода очередного кадра возвращаться в цикл событий, обеспечивая нормальную работу Windows. Если все это представляется вам слишком запутанным и непонятным, не волнуйтесь — дальше будет еще хуже.

При работе Game_Main() выполняется основная логика игры и в отдельном буфере создается новый кадр игры, который затем выводится на экран вызовом DD_Flip(). Еще раз посмотрите исходный текст игры и проследите все состояния игры, их функции и переходы между ними. Для того чтобы сыграть, просто запустите на выполнение файл FREAKOUT.EXE. Игрой управляют три клавиши:

<←> — перемещение ракетки влево;
<→> — перемещение ракетки вправо;
<Esc> — выход из игры.

Не забывайте, что при потере мяча с вас снимается 100 очков, так что будьте внимательны!

Когда вы разберетесь с игрой и ее кодом, попробуйте вносить в текст небольшие изменения. Вы можете изменить цвет фона, сделать ракетку с изменяющейся длиной, добавить мячи и даже изменить убогое звуковое сопровождение (которое сейчас ограничено вызовом `MessageBeep()`).

Резюме

В этой главе я хотел всего лишь дать представление о том, о чем будет идти речь в книге, а также показать пример завершенной игры, так как это сразу вызывает множество вопросов, над которыми следует задуматься читателю.

Перед тем как перейти к чтению следующей главы, убедитесь, что процесс компиляции FreakOut не вызывает у вас никаких сложностей. Если это не так, то, прежде чем продолжить чтение книги, прочитайте документацию к вашему компилятору и решите все связанные с ним вопросы. Я подожду.

ГЛАВА 2

Модель программирования Windows

Программирование в Windows напоминает поход к стоматологу: умом понимаешь всю пользу этого действия, но выполнять его крайне неприятно. Или я не прав? В этой главе я познакомлю вас с основами программирования в Windows, базируясь на собственной методологии; другими словами, обучу вас как можно проще. Не могу обещать, что после чтения этой главы вы будете с радостью предвкушать посещение стоматолога, но смею вас уверить, что программировать в Windows вам понравится. Итак, в этой главе рассматриваются следующие вопросы.

- История Windows
- Архитектура Windows
- Классы окон
- Создание окон
- Обработчики событий
- Программирование, управляемое событиями, и циклы событий
- Открытие многих окон

Происхождение Windows

Начнем рассмотрение с того, как Windows стала Windows и каковы ее взаимосвязи с миром разработки игр.

Ранние версии Windows

Все началось с Windows 1.0. Это была первая попытка Microsoft создать коммерческую оконную операционную систему, и вся она оказалась одной большой ошибкой. Windows 1.0 базировалась на DOS (большая ошибка), не была многозадачной, работала чересчур медленно и выглядела слишком плохо. (Пожалуй, основной причиной ее неуспеха был именно плохой внешний вид.) Но если говорить серьезно, то главная проблема заключалась в том, что Windows требовалось более мощное аппаратное обеспечение, графические и звуковые возможности, чем те, которые могли дать машины с процессором 80286 (или, что еще хуже, 8086).

Тем не менее Microsoft двигалась вперед, и вскоре свет увидела следующая версия — Windows 2.0. Я помню свою работу в Software Publishing Corporation, когда мы получили бета-версию Windows 2.0. Зал для совещаний был забит сотрудниками; присутствовал даже президент компании (с неизменным коктейлем в руке). Мы запустили демоверсию Windows 2.0 и загрузили несколько приложений, и все казалось вполне работоспособным. Но к тому времени уже увидел свет IBM Presentation Manager (PM), который не только гораздо лучше выглядел, но и работал под OS/2, которая была полноценной операционной системой, существенно превосходящей во всех отношениях Windows 2.0 (бывшей не более чем надстройкой над DOS). Так что на собрании был вынесен следующий вердикт: "Неплохо, но для разработки не годится. Остаемся с DOS, и где мой коктейль?"

Windows 3.x

В 1990 году наконец-то появилась Windows 3.0, которая выглядела вполне прилично. Конечно, она до боли напоминала Mac OS, но кого это волновало? (В конце концов, настоящие программисты терпеть не могут Mac.) Наконец разработчики программного обеспечения могли создавать привлекательные приложения для PC и начать перенос программ для DOS в Windows. Это была поворотная точка, когда PC начал побеждать Mac на рынке бизнес-приложений, а затем и на рынке настольных издательских систем (при том, что Apple выпускала новые версии аппаратного обеспечения каждые пять минут).

Несмотря на приличную работу Windows 3.0, в ней осталось немало проблем, ошибок и прочих неприятностей. Появление Windows 3.0 оказалось принципиальным скачком в развитии технологии, так что определенные неприятности были не такими уж неожиданными. Для исправления этих проблем Microsoft выпустила Windows 3.1. Отдел маркетинга Microsoft хотел дать ей имя Windows 4.0, но потом было решено сделать ее Windows 3.1, поскольку в систему было внесено не так уж много новшеств, чтобы изменять старший номер версии.

Одним из новшеств Windows 3.1 стала поддержка видео и звука, другим (в версии 3.11 — Windows for Workgroups) — поддержка работы в сети. Главная же проблема Windows 3.1 оставалась в том, что это было, по сути, приложение DOS.

Windows 95

В 1995 году Microsoft сумела создать настоящую 32-битовую, многозадачную, многопоточную операционную систему (несмотря на то что в ней оставался немалый кусок 16-битового кода). Конечно, была еще линия Windows NT 3.0, но поскольку это операционная система не для среднего пользователя, речь о ней здесь идти не будет.

После выпуска Windows 95 даже мне захотелось программировать для нее. Программирование для Windows 1.0, 2.0, 3.0 и 3.1 было для меня ненавистным занятием (впрочем, эта ненависть уменьшалась с каждой новой версией). Появление же Windows 95 изменило мое отношение к этому занятию, как и у множества других программистов. Наконец у нас было то, в чем мы так долго нуждались.

Выход Windows 95 изменил компьютерный бизнес. Правда, многие компании и сегодня используют Windows 3.1 (вы в состоянии в это поверить?), но Windows 95 сделала PC компьютером для приложений *любого* типа, за исключением игр. Да, DOS все еще оставалась непревзойденной системой для программистов игр, хотя даже они понимали, что переход игр под Windows не более чем вопрос времени.

В 1996 году Microsoft выпустила комплект Game SDK, который, по сути, можно считать первой версией DirectX. Эта технология работала только в Windows 95, но была слишком медленной, чтобы состязаться с такими играми для DOS, как DOOM или Duke Nukem. Разработчики игр продолжали работать под DOS32, хотя и знали, что скоро наступит время, когда технология DirectX станет достаточно быстрой для разработки игр для PC.

Начиная с версии 3.0, DirectX сравнялась по скорости с работой DOS32, а в версии 5.0, по сути, стали реальностью все заявленные возможности этой технологии. На момент написания этой книги Microsoft работала над версией DirectX 9.0; доступна для использования версия 8.1. Однако подробнее DirectX рассматривается в главе 5, "Основы DirectX и COM", а пока упомянем, что, по сути, единственным способом использования PC для игр является связка Win32/DirectX.

Windows 98

В средине 1998 года была выпущена новая версия операционной системы — Windows 98. Это скорее эволюционный, чем революционный шаг в развитии технологий, но от этого он не становится менее важным. Windows 98 — это полностью 32-битовая операционная система с поддержкой всего, что только можно себе вообразить, открытая для всевозможных расширений. Она обладает встроенной поддержкой сетей и Internet, мультимедиа, 3D-графики, DirectX.

Windows 98 существенно надежнее в работе по сравнению с Windows 95. Конечно, и Windows 98 может взбрыкнуть, но с ней это случается гораздо реже, чем с Windows 95. Кроме того, Plug and Play в Windows 98 работает на самом деле и работает неплохо.

Windows ME

В конце 1999 — начале 2000 года увидела свет версия Windows ME (Millennium). Она, по сути, представляет собой Windows 98 с увеличенной степенью интегрированности поддержки сети и мультимедиа. Ее появление — в большей степени коммерческий ход, чем реальная техническая необходимость. Windows ME более капризна, в ней плохо работает ряд приложений, ее "не любит" некоторое аппаратное обеспечение, так что эта операционная система хорошо работает, как правило, только на очень новых машинах, в то время как у Windows 98 обычно не возникает проблем при работе на компьютерах, выпущенных еще в 1995 году. Однако на новых машинах Windows ME — достаточно надежная операционная система для игр.

Windows XP

Эта операционная система сразу понравилась мне своим внешним видом, можно сказать — своей эротичностью. Она выглядит как Windows 98 или ME, в то же время обладая надежностью Windows 2000 или NT. Это был очередной шаг в области операционных систем для средних пользователей. Но наряду со всеми прелестями имеется и обратная сторона медали. Microsoft годами предупреждала производителей аппаратного и программного обеспечения о том, что не надо "химичить" и использовать в своем коде не совсем корректные способы работы с аппаратным обеспечением. В результате в истинно 32-битовой Win32-совместимой операционной системе просто не запускается ряд приложений, нарушающих правила Microsoft. В конечном счете это даже неплохо, так как

производители будут вынуждены откорректировать свой код (который и был одной из причин нестабильности работы Windows 95/98).

Ну а пока, чтобы не потерять клиентов из-за такой несовместимости, XP имеет два варианта помощи пользователям в данной ситуации. Во-первых, операционная система постоянно обновляется посредством Microsoft update и пользователь немедленно получает исправленное программное обеспечение. Во-вторых, для работы с проблемным программным обеспечением в XP имеется возможность включения так называемого режима совместимости (с несовместимым программным обеспечением; по сути, его работа сводится к тому, чтобы восстановить все ошибки, — и тогда программы заработают :-)).

Windows NT/2000

Теперь мы можем поговорить и о Windows NT. Во время написания этой книги текущей была версия 4.0 release 5, которая была переименована в Windows 2000 и, по сути, представляет собой NT 5.0. Как я уже говорил, линии 9X и NT сходятся в одной точке — Windows XP, предназначенной для широкого круга пользователей. Windows 2000 попросту существенно строже Windows XP, так что многие производители игр предпочитают разрабатывать их под Windows 2000, а затем выпускать для Windows 9X/ME/XP. Windows 2000 обладает полной поддержкой Plug and Play наряду с Win32/DirectX, так что приложения, написанные для Windows 9X с DirectX, будут нормально работать в Windows 2000.

Итак, какой из всего сказанного следует вывод? Если вы пишете приложение Win32 с использованием DirectX (или без него), то такое приложение будет корректно работать в Windows 95, 98, ME, XP и 2000. Следовательно, все изложенное в этой книге применимо для целого ряда операционных систем семейства Windows. И не забывайте о Windows CE 3.0/Pocket PC 2000 — DirectX и подмножество Win32 работают и в этой системе.

Базовая архитектура Windows: Win9X/NT

Windows, в отличие от DOS, представляет собой многозадачную операционную систему, созданную для одновременной работы ряда приложений и/или меньших процессов с максимальными возможностями использования аппаратного обеспечения. Это означает, что Windows является разделяемой средой: ни одно приложение не может получить в свое распоряжение всю систему целиком. Хотя Windows 95, 98, ME, XP и 2000/NT похожи, они имеют ряд технических отличий. Однако в этой книге рассматриваются общие черты, а не отличия, так что делать большую драму из различия операционных систем не стоит. Итак, приступим!

Многозадачность и многопоточность

Как уже отмечалось, Windows позволяет выполняться нескольким приложениям одновременно, при этом каждое приложение по очереди получает малый отрезок времени для выполнения, после чего наступает черед другого приложения. Как показано на рис. 2.1, процессор совместно используется несколькими выполняющимися процессами. Точное определение, какой именно процесс будет выполняться следующим и какое процессорное время выделяется каждому из приложений, — задача *планировщика*.

Планировщик может быть очень простым, обеспечивающим выполнение каждого из процессов одинаковое количество миллисекунд, а может быть и очень сложным, работающим с учетом различных уровней приоритета приложений и вытесняющим низкоприоритетные приложения. В Win 9X/NT используется вытесняющий планировщик, работающий с учетом приоритетов. Это означает, что одни приложения могут получить больше процессорного времени, чем другие.

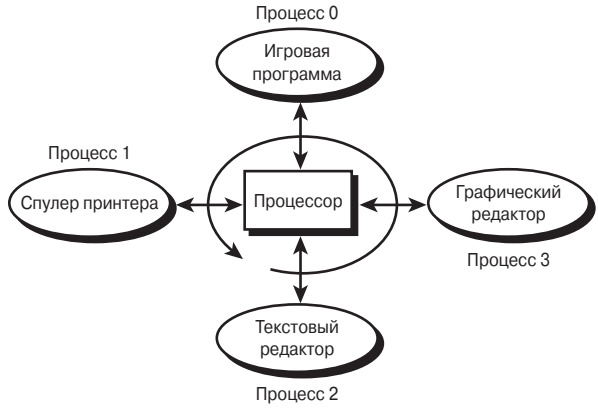


Рис. 2.1. Многозадачность в однопроцессорной системе

Однако беспокоиться о работе планировщика не стоит, если только вы не разработчик операционной системы или программы, работающей в реальном времени. В большинстве случаев Windows сама запустит и спланирует приложение, и с вашей стороны для этого не требуется никаких специальных действий.

Познакомившись с Windows поближе, мы увидим, что это не только многозадачная, но и *многопоточная* операционная система. Это означает, что в действительности программы состоят из ряда более простых *потоков выполнения*. Выполнение этих потоков планируется так же, как и выполнение более мощных процессов, таких, как программы. Вероятно, в настоящий момент на вашем компьютере работает от 30 до 50 потоков, выполняющих разные задачи. Итак, в Windows единая программа может состоять из одного или нескольких потоков выполнения.

На рис. 2.2 схематически показана многопоточность в Windows. Как видите, каждая программа в действительности состоит, в дополнение к основному потоку, из нескольких рабочих потоков.

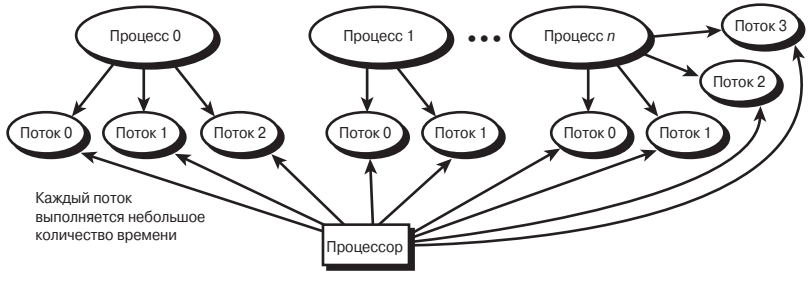


Рис. 2.2. Многопоточность в Windows

Получение информации о потоках

Для развлечения посмотрим, сколько потоков выполняется на вашей машине в настоящий момент. Нажмите <Ctrl+Alt+Delete> на компьютере под управлением Windows для вызова Active Program Task Manager и посмотрите, чему равно количество выполняющихся в системе потоков (или процессов). Это не совсем та величина, которая нас интересует, но весьма близкая к ней. Нас интересует приложение, способное сообщить

реальное количество выполняющихся процессов. Для этого подходит множество условно бесплатных и коммерческих программ, но они нас не интересуют, поскольку в Windows есть встроенное средство для получения этой информации.

В каталоге Windows (в большинстве случаев это каталог \WINDOWS) можно обнаружить программу SYSMON.EXE (она не включена в установку Windows по умолчанию, так что при ее отсутствии просто добавьте ее в систему посредством Control Panel⇨Add/Remove Programs⇨ System Tools) или, в Windows NT, PERFMON.EXE. На рис. 2.3 показана копия экрана SYSMON.EXE на моей машине под управлением Windows 98. Как видите, кроме количества потоков, данная программа предоставляет и другую важную информацию, такую, как использование памяти и загрузка процессора. Я часто использую эту программу, чтобы отслеживать, что происходит при работе создаваемых мною программ.

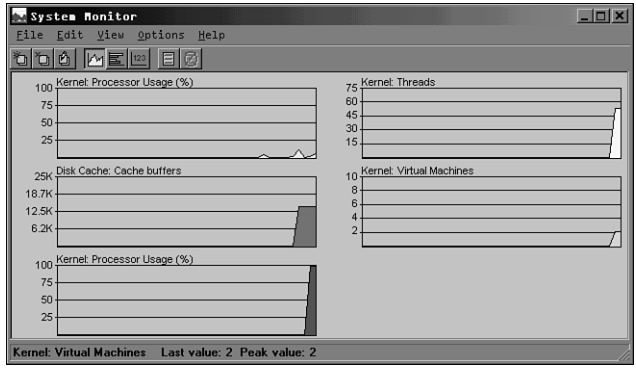


Рис. 2.3. Окно SYSMON

А теперь о приятном: вы можете сами управлять созданием потоков в своих программах. Это одна из наиболее увлекательных возможностей при программировании игр — мы можем создать столько потоков, сколько нам потребуется для выполнения различных задач в дополнение к основному процессу игры.

НА ЗАМЕТКУ

В Windows 98/NT введен новый тип объекта выполнения — нить (fiber), который еще проще, чем поток.

Вот основное отличие игр для Windows от игр для DOS. Поскольку DOS — однозадачная операционная система, в ней после запуска программы на выполнение больше ничего другого выполняться не может (не считая время от времени вызываемых обработчиков прерываний). Следовательно, если вы хотите добиться многозадачности или многопоточности в DOS, вам придется эмулировать ее самостоятельно (см., например, книгу *Teach Yourself Game Programming in 21 Days*, где описано многозадачное ядро на основе DOS). И это именно то, чем многие годы занимались программисты игр. Конечно, эмуляция многозадачности и многопоточности никогда не будет такой же надежной, как реальная многозадачность и многопоточность в поддерживающей их операционной системе, но для отдельной игры такой эмуляции вполне достаточно.

Перед тем как перейти к программированию в Windows, хочу упомянуть еще одну деталь. Вы можете подумать, что Windows — волшебная операционная система, поскольку позволяет одновременно решать несколько задач и выполнять несколько программ. Но это *не* так. Если в системе только один процессор, то одновременно может выполняться только один поток, программа или другая единица выполнения. Windows просто переключается между ними так быстро, что создается иллюзия одновременной работы не-

скольких программ. Если же в системе несколько процессоров, то несколько задач могут выполняться действительно одновременно. Например, у меня есть компьютер с двумя процессорами Pentium II 400MHz, работающий под управлением Windows 2000. В этой системе действительно возможно одновременное выполнение двух потоков инструкций.

Я думаю, что в ближайшем будущем следует ожидать новую архитектуру микропроцессоров для персональных компьютеров, которая обеспечит одновременное выполнение нескольких потоков как часть конструкции процессора. Например, процессор Pentium имеет два модуля выполнения — U- и V-каналы. Следовательно, он может одновременно выполнять две инструкции, однако эти инструкции всегда из одного и того же потока. Аналогично, процессоры Pentium II, III, IV также могут выполнять несколько инструкций одновременно, но только из одного и того же потока.

Модель событий

Windows является многозадачной и многопоточной операционной системой, но при этом она остается операционной системой, *управляемой событиями* (event-driven). В отличие от программ DOS, большинство программ Windows попросту ждут, пока пользователь не сделает что-то, что запустит событие, в ответ на которое Windows предпримет некоторые действия. На рис. 2.4 вы можете рассмотреть работу этой системы. Здесь изображены несколько окон приложений, каждое из которых посылает свои события или сообщения Windows для последующей обработки. Windows выполняет обработку определенных сообщений, но большинство из них передаются для обработки вашему приложению.

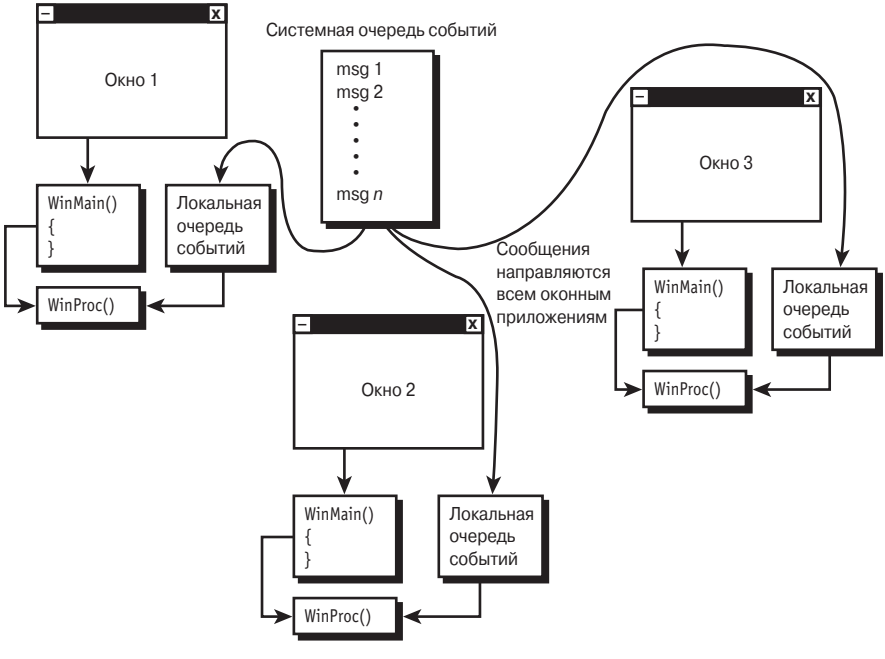


Рис. 2.4. Обработка событий в Windows

Хорошая новость состоит в том, что вам нет необходимости беспокоиться о других работающих приложениях — Windows сама разберется с ними. Все, что вы должны сделать, — это позаботиться о вашем собственном приложении и обработке сообщений для вашего окна (окон). Ранее, в Windows 3.0/3.1, это было не так. Эти версии Windows не были истинно мно-

гозадачными операционными системами, и каждое приложение должно было передать управление следующему. Это означало, что если находилось приложение, которое ухитрялось надолго захватить систему, другие приложения не могли ничего с этим поделать.

Теперь о концепциях операционной системы вам известно все, что я намеревался рассказать. К счастью, Windows настолько хорошо подходит для написания игр, что вам не нужно заботиться о планировании — от вас требуется лишь код игры.

Далее в этой главе вы встретитесь с реальным программированием и увидите, насколько простое это занятие. Но (всегда это "но"!) пока что вам следует познакомиться с некоторыми соглашениями, используемыми программистами Microsoft. Применяя их, вы никогда не запутаетесь среди имен функций и переменных.

Венгерская нотация

Если бы вы руководили большой компанией типа Microsoft с тысячами программистов, работающими над различными проектами, вы бы непременно пришли к необходимости введения стандарта написания кода, в противном случае хаос был бы неизбежен. Так и была рождена спецификация написания кода в Microsoft. Ее автор — человек по имени Чарльз Симони (Charles Simonyi). Все API, интерфейсы, технические статьи и прочее используют эти соглашения.

Обычно их называют *венгерской нотацией*. Почему? Вряд ли когда-нибудь мы точно узнаем об этом, да это и неважно — сейчас перед нами другая цель: научиться читать код Microsoft.

Венгерская нотация состоит из ряда соглашений по именованию переменных, функций, типов, констант, классов и параметров функций. В табл. 2.1 содержатся все префиксы, используемые при этом именовании.

Таблица 2.1. Спецификация префиксов венгерской нотации

Префикс	*Тип данных (базовый тип)*
c	char
by	BYTE (unsigned char)
n	short int, int
i	int
x, y	short (в качестве координат x и y)
cx, cy	short (в качестве длины вдоль координат x и y)
b	BOOL (int)
w	UINT (unsigned int) или WORD (unsigned short int)
l	LONG (long)
dw	DWORD (unsigned long)
fn	Указатель на функцию
s	Строка
sz, str	Строка, завершающаяся нулевым байтом
lp	32-битовый указатель
h	Дескриптор (используется для обращения к объектам Windows)
msg	Сообщение

Именование переменных

При использовании венгерской нотации переменные имеют префиксы, указанные в табл. 2.1. Кроме того, если имя переменной состоит из одного или нескольких слов, то каждое из них записывается с прописной буквы. Вот несколько примеров:

```
char  *szFileName;      // Строка с завершающим нулем
int   *lpiData;         // 32-битовый указатель на int
BOOL  bSemaphore;       // Логическое значение
DWORD dwMaxCount;  // 32-битовое беззнаковое целое
```

Хотя специальное обозначение для локальных переменных мне неизвестно, для глобальных обычно используется префикс g_ (или просто g):

```
int  g_iXPos;       // Глобальная позиция по координате x
int  g_iTimer;      // Глобальный таймер
char *g_szString;   // Глобальная строка с завершающим нулем
```

Именование функций

Функции именуются так же, как и переменные, но только без префиксов. Другими словами, в именах функций составляющие их слова пишутся с прописных букв, и не более того.

```
int  PlotPixel(int ix, int iy, int ic);
void *MemScan(char *szString);
```

Использование подчеркиваний считается неверным. Например, следующая функция не соответствует венгерской нотации:

```
int Get_Pixel(int ix, int iy);
```

Именование типов и констант

Имена всех типов и констант состоят только из прописных букв, но при этом вы можете использовать в именах подчеркивание:

```
const LONG NUM_SECTORS = 100;  // Константа в стиле C++
#define  MAX_CELLS  64         // Константа в стиле C
#define  POWERUNIT  100        // Константа в стиле C
typedef unsigned char UCHAR;   // Пользовательский тип
```

Здесь нет ничего необычного — обыкновенные стандартные определения. Хотя большинство программистов Microsoft не используют подчеркивания, я предпочитаю пользоваться ими с тем, чтобы имена были более удобочитаемы.

СОВЕТ

> В C++ ключевое слово const имеет несколько значений, но в приведенном выше коде оно использовано просто для создания константной переменной. Этот способ похож на использование макроопределения #define, но при этом содержит информацию о типе. Использование const позволяет, таким образом, обеспечить проверку типов в процессе компиляции и выполнить необходимые преобразования.

Именование классов

Соглашение об именовании классов может вас несколько удивить, но я видел множество программистов, настолько привыкших к нему, что используют его в повседневной работе. Просто все классы C++ должны иметь имена с префиксом, состоящим из одной прописной буквы C.

```
class CVector
{
public:
   CVector(){ ix = iy = iz = imagnitude = 0; }
   CVector(int x, int y, int z){ ix = x; iy = y; iz = z; }

.
.

private:
   int ix, iy, iz;  // Направление вектора
   int imagnitude;  // Размер вектора
};
```

Именование параметров

Имена параметров функций следуют тем же соглашениям, что и обычные переменные, хотя это и не обязательно. Например, вы можете увидеть объявление функции, выглядящее так:

```
UCHAR GetPixel(int x, int y);
```

При использовании венгерской нотации это объявление должно выглядеть следующим образом:

```
UCHAR GetPixel(int ix, int iy);
```

Кроме того, в объявлении функции имена параметров могут не указываться вовсе — указываются только их типы:

```
UCHAR GetPixel(int, int);
```

Конечно, такое допустимо только в объявлении функции; в определении функции должны использоваться не только типы, но и имена переменных.

НА ЗАМЕТКУ

Знакомство с венгерской нотацией еще не означает, что вы в обязательном порядке должны ее использовать. Например, я программирую более 20 лет и не намерен менять свой стиль ради кого бы то ни было. Таким образом, код в этой книге будет использовать венгерскую нотацию при рассмотрении функций Win32 API, но в других местах я буду пользоваться собственным стилем. Так, например, я не собираюсь начинать каждое слово в именах переменных с прописной буквы, да и подчеркиваниями я пользуюсь нередко.

Простейшая программа Windows

Теперь, после краткого обзора операционных систем семейства Windows и знакомства с их свойствами, приступим к реальному программированию и начнем с программы, которая, как это принято для первой программы во вновь изучаемом языке, выводит на экран строку “Hello, World”. В листинге 2.1 приведена такая программа для операционной системы DOS.

Листинг 2.1. Программа “Hello, World” для DOS

```
// DEMO2_1.CPP - стандартная версия
#include <stdio.h>

// главная входная точка для всех стандартных
// программ для DOS и консольных программ
```

```
void main(void)
{
   printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // main
```

СОВЕТ

> Кстати, если вы хотите скомпилировать DEMO2_1.CPP, то можете с помощью компиляторов VC++ или Borland создать так называемое *консольное приложение*, которое очень похоже на приложения DOS, но только являющиеся 32-битовыми. Такое приложение работает лишь в текстовом режиме, но зато отлично подходит для отработки идей и алгоритмов. Для этого убедитесь, что в компиляторе установлена опция создания консольного приложения, а не приложения Win32.

Для компиляции этой программы следуйте инструкции.

1. Создайте новый проект консольного приложения и включите в него файл DEMO2_1.CPP из каталога \T3DGameR1\Source\T3DCHAP02\ на прилагаемом компакт-диске.
2. Скомпилируйте и скомпонуйте программу.
3. Запустите ее (или запустите готовую версию DEMO2_1.EXE, имеющуюся на компакт-диске).

Все начинается с WinMain()

Как упоминалось ранее, все программы Windows начинают выполнение с функции WinMain(), которая представляет собой эквивалент main() в обычных программах DOS. Все, что вы хотите сделать в своей программе, делается в функции WinMain(): создается окно, начинается обработка событий и осуществляется вывод на экран. Другими словами, здесь вы можете вызвать любую из сотен (или тысяч?) функций Win32 API. Что мы сейчас и сделаем.

Сейчас я хочу написать программку, выводящую текст в небольшом окошке сообщений. Это именно то, что происходит при вызове одной из функций Win32 API — MessageBox(). В листинге 2.2 приведен полный текст программы для Windows, создающей и выводящей окно сообщения, которое можно перемещать по экрану и закрыть.

Листинг 2.2. Ваша первая программа для Windows

```
// DEMO2_2.CPP - простое окно сообщений
#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // Главный заголовочный файл
#include <windowsx.h> // Набор макросов

// Главная входная точка Windows-программ
int WINAPI WinMain(HINSTANCE hinstance,
            HINSTANCE hprevinstance,
            LPSTR     lpcmdline,
            int       ncmdshow)
{
// Вызов MessageBox с передачей нулевого
// дескриптора родительского окна
MessageBox(NULL, "THERE CAN BE ONLY ONE!!!",
           "MY FIRST WINDOWS PROGRAM",
           MB_OK | MB_ICONEXCLAMATION);
```

```
// Выход из программы
return(0);

} // WinMain
```

Для компиляции этой программы следуйте инструкции.

1. Создайте новый проект приложения Win32 и включите в него файл DEMO2_2.CPP из каталога \T3DGameR1\Source\T3DCHAP02\ на прилагаемом компакт-диске.
2. Скомпилируйте и скомпонуйте программу.
3. Запустите ее (или запустите готовую версию DEMO2_2.EXE, имеющуюся на компакт-диске).

А вы, наверное, думали, что не бывает программ для Windows менее чем с парой сотен строк? Но, как видите, все гораздо проще, и вы можете легко скомпилировать и запустить простенькую программу, которая показана на рис. 2.5.

Рис. 2.5. Работа программы DEMO2_2.EXE

Разбор программы

Теперь, когда у нас есть полная программа для Windows, разберем ее строка за строкой и посмотрим, как она работает. Самая первая строка программы

```
#define WIN32_LEAN_AND_MEAN
```

Эта строка заслуживает отдельного пояснения. Имеется два способа создания программ для Windows: с использованием Microsoft Foundation Classes (MFC) или с помощью Software Development Kit (SDK). Способ MFC более сложный, полностью базируется на C++ и классах и раз в 10 мощнее, чем необходимо для разработки игр. SDK же существенно проще, легче в изучении и использует простой C, следовательно, именно SDK я и буду использовать в этой книге.

Строка #define WIN32_LEAN_AND_MEAN служит для того, чтобы указать компилятору (через включаемые файлы), что включать ненужные в данной программе файлы с объявлениями MFC не следует.

```
#include <windows.h>   // Главный заголовочный файл
#include <windowsx.h> // Набор макросов
```

Первая директива действительно включает все заголовочные файлы Windows. Их чрезвычайно много, и первую директиву можно рассмотреть как экономящую ваши силы и время и позволяющую обойтись без отдельного включения каждого необходимого файла. Вторая директива включает заголовочный файл, содержащий ряд важных макросов и констант, которые упрощают программирование.

Теперь перейдем к самому основному — главной входной точке всех приложений Windows, функции WinMain().

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

Сначала вы должны обратить внимание на странный декларатор WINAPI. Он эквивалентен декларатору PASCAL, который заставляет осуществлять передачу параметров слева направо, а не справа налево, как в случае декларатора по умолчанию CDECL. Вы обязаны использовать декларатор WINAPI для функции WinMain(), в противном случае загрузочный код неверно передаст параметры данной функции.

Параметры WinMain()

Теперь рассмотрим каждый параметр функции WinMain().

- hinstance. Этот параметр является дескриптором, который Windows генерирует для вашего приложения и который в дальнейшем применяется для отслеживания использования ресурсов.
- hprevinstance. Этот параметр больше не используется, но ранее он отслеживал экземпляр предыдущего приложения (иными словами, приложения, которое запускает текущее).
- lpcmdline. Это строка с завершающим нулевым символом, аналогичная параметрам командной строки стандартной функции main(int argc, char **argv), с тем лишь отличием, что здесь нет отдельного параметра типа argc, который бы указывал количество параметров в командной строке. Например, при создании Windows-приложения TEST.EXE и запуске его с параметрами TEST.EXE one two three строка lpcmdline будет иметь вид lpcmdline = "one two three". Заметим, что само имя исполняемого файла *не* является частью командной строки.
- ncmdshow. Этот последний параметр представляет собой целое число, которое передается приложению при запуске и указывает, как именно это приложение должно быть открыто. Таким образом, пользователь получает определенную возможность управления запуском приложения. Конечно, как программист, вы можете просто проигнорировать данный параметр, но, если хотите, можете им воспользоваться. (Вы передаете этот параметр функции ShowWindow(), но можете воспользоваться им и в других целях.) В табл. 2.2 перечислены наиболее часто использующиеся значения данного параметра.

Таблица 2.2. Значения параметра ncmdshow

Значение	*Функция*
SW_SHOWNORMAL	Активизирует и показывает окно. Если окно находится в минимизированном или максимизированном состоянии, Windows восстанавливает его исходный размер и положение. Приложение должно указать этот флаг при первом выводе окна
SW_SHOW	Активизирует окно и выводит его с текущим положением и размером
SW_HIDE	Скрывает окно и активизирует другое окно
SW_MAXIMIZE	Максимизирует указанное окно
SW_MINIMIZE	Минимизирует указанное окно и активизирует следующее окно в Z-порядке
SW_RESTORE	Активизирует и выводит окно. Если окно минимизировано или максимизировано, Windows восстанавливает его исходный размер и положение. Приложение должно указать этот флаг при восстановлении минимизированного окна

Окончание табл. 2.2

Значение	*Функция*
SW_SHOWMAXIMIZED	Активизирует окно и выводит его как максимизированное
SW_SHOWMINIMIZED	Активизирует окно и выводит его как минимизированное
SW_SHOWMINNOACTIVE	Выводит окно как минимизированное. Активное окно остается при этом активным
SW_SHOWNA	Выводит окно в его текущем состоянии. Активное окно остается при этом активным
SW_SHOWNOACTIVATE	Выводит окно с последним размером и в последнем положении. Активное окно остается при этом активным

Как видите, существует множество вариантов значения ncmdshow (многие из которых в функции WinMain() не имеют смысла). В реальной ситуации большинство этих значений никогда не передаются функции WinMain(). Затем вы можете использовать их в другой функции, ShowWindow(), которая выводит созданное окно, но об этом речь идет несколько позже.

Главное, на что я хотел обратить ваше внимание, — это на то, что у Windows есть огромное количество различных опций, флагов и т.п., которые никогда не используются. Что касается ncmdshow, то в действительности в 99% случаев используются значения SW_SHOW, SW_SHOWNORMAL и SW_HIDE, но это не отменяет необходимость знать и об оставшемся проценте!

Выбор окна сообщения

Теперь поговорим о вызове функции MessageBox(). Это функция Win32 API, которая используется для вывода сообщения с различными пиктограммами и одной или двумя кнопками. Эта задача очень часто встречается в повседневной работе каждого программиста, так что наличие специализированной функции существенно экономит время.

Функции MessageBox() достаточно, чтобы вывести на экран окно, задать вопрос и дождаться ответа пользователя. Вот прототип этой функции:

```
int MessageBox(HWND   hwnd,    // Дескриптор окна-владельца
       LPCTSTR lptext,         // Выводимый текст
       LPCTSTR lpcaption,      // Заголовок окна
       UINT   utype);          // Стиль окна сообщения
```

Перечислим параметры функции MessageBox().

- hwnd. Дескриптор окна, к которому должно быть присоединено окно сообщения. Пока что мы не разбирались с дескрипторами окон, а значит, можно рассматривать это окно как родительское для выводимого окна сообщений. В нашем примере это значение равно NULL, так что в качестве родительского окна выступает десктоп.
- lptext. Это строка с завершающим нулевым символом, в которой содержится выводимый в окне текст.
- lpcaption. Это строка с завершающим нулевым символом, в которой содержится заголовок окна сообщений.
- utype. Едва ли не самый важный параметр, который определяет вид выводимого окна сообщений. В табл. 2.3 приведен неполный список различных значений этого параметра.

Таблица 2.3. Опции MessageBox()

Флаг	*Описание*
Определение общего вида окна сообщений	
MB_OK	Окно сообщений содержит одну кнопку OK. Это вид окна по умолчанию
MB_OKCANCEL	Окно сообщений содержит две кнопки: OK и Cancel
MB_RETRYCANCEL	Окно сообщений содержит две кнопки: Retry и Cancel
MB_YESNO	Окно сообщений содержит две кнопки: Yes и No
MB_YESNOCANCEL	Окно сообщений содержит три кнопки: Yes, No и Cancel
MB_ABORTRETRYIGNORE	Окно сообщений содержит три кнопки: Abort, Retry и Ignore
Определение вида пиктограммы	
MB_ICONEXCLAMATION	В окне используется пиктограмма с восклицательным знаком
MB_ICONINFORMATION	В окне используется пиктограмма с буквой i в кружке
MB_ICONQUESTION	В окне используется пиктограмма с вопросительным знаком
MB_ICONSTOP	В окне используется пиктограмма, представляющая собой знак “Стоп”
Какая из кнопок является кнопкой по умолчанию	
MB_DEFAULTBUTTON*n*	Значение *n* представляет собой число (1...4), указывающее номер кнопки (слева направо), являющейся кнопкой по умолчанию

Примечание. Имеется ряд дополнительных флагов уровня операционной системы, но мы их не рассматриваем. При желании вы можете найти информацию о них в справочной системе Win32 SDK.

Значения из табл. 2.3 могут быть объединены посредством побитового ИЛИ. Обычно при этом используется по одному значению из каждой группы.

Разумеется, функция MessageBox() возвращает значение, позволяющее вам узнать, как отреагировал на появившееся окно сообщения пользователь. В нашем случае это совершенно неважно, но, вообще говоря, вы можете захотеть узнать, какая же из кнопок была нажата пользователем, ответил ли он “да” или “нет” и т.д. В табл. 2.4 перечислены возможные возвращаемые значения данной функции.

Таблица 2.4. Возвращаемые значения функции MessageBox()

Значение	*Выбранная кнопка*
IDABORT	Abort
IDCANCEL	Cancel
IDIGNORE	Ignore
IDNO	No
IDOK	OK
IDRETRY	Retry
IDYES	Yes

На этом мы завершили разбор нашей программы.

Если вы хотите не только выводить окно сообщения, но и слышать при этом звуковое сопровождение, можно воспользоваться для этого функцией MessageBeep(), также имеющейся в Win32 SDK. По простоте использования она не уступает функции MessageBox():

```
BOOL MessageBeep(UINT utype); // Вывод звука
```

СОВЕТ

Было бы неплохо, если бы вы немного "поиграли" с программой: внесли в нее изменения, перекомпилировали ее с различными опциями компилятора, включая разные варианты оптимизации, прошли ее пошагово в отладчике — словом, приобрели навыки работы с инструментарием программиста.

Выводимый функцией звук зависит от значения ее параметра; возможные значения показаны в табл. 2.5.

Таблица 2.5. Вывод звука функцией `MessageBeep()`

Значение параметра	*Звук*
`MB_ICONASTERISK`	Воспроизведение звука, соответствующего пиктограмме информации
`MB_ICONEXCLAMATION`	Воспроизведение звука, соответствующего пиктограмме с восклицательным знаком
`MB_ICONHAND`	Воспроизведение звука, соответствующего пиктограмме "стоп"
`MB_ICONQUESTION`	Воспроизведение звука, соответствующего пиктограмме с вопросительным знаком
`MB_OK`	Воспроизведение системного звука по умолчанию
`0xFFFFFFFF`	Вывод стандартного системного звука на встроенный динамик

Примечание. Очень интересные результаты можно получить на компьютере с установленным пакетом MS-Plus Theme.

Теперь вы видите, какая это замечательная штука — Win32 API? Имеются буквально сотни функций, которые вы можете использовать в своих программах.

Итак, еще раз резюмируем все, что вы узнали о программировании в Windows к настоящему моменту. Первое: Windows является многозадачной и многопоточной операционной системой, так что в ней одновременно может выполняться несколько приложений. Однако нам не следует специально об этом заботиться и предпринимать какие-то особые шаги. Второе: Windows представляет собой систему, управляемую событиями. Это означает, что мы должны обрабатывать события (пока что мы не знаем, как именно это делается) и отвечать на них. И наконец, все программы Windows начинаются с функции `WinMain()`, которая имеет немного больше параметров, чем стандартная функция `main()`.

Теперь самое время приступить к написанию реального приложения Windows.

Написание реального Windows-приложения

Несмотря на то что наша конечная цель — написание трехмерных игр для Windows, нам не так уж много требуется знать о программировании в Windows. Все, что нужно, — это базовая программа Windows, которая открывает окно, обрабатывает сообщения и запускает основной цикл игры. Исходя из этого, моя цель в данном разделе — показать вам, как создается такая простая программа.

Главное в любой Windows-программе — это открытие окна. *Окно* представляет собой не что иное, как рабочее пространство для вывода информации, такой, как текст и графика, так что пользователь может взаимодействовать с ней. Для создания полнофункциональной Windows-программы вы должны выполнить ряд действий.

1. Создать класс Windows.
2. Создать обработчик событий.

3. Зарегистрировать класс в Windows.
4. Создать окно с предварительно созданным классом.
5. Создать главный цикл событий, который получает и передает сообщения Windows обработчику событий.

Рассмотрим каждый шаг детальнее.

Класс Windows

Windows представляет собой объектно-ориентированную операционную систему, так что многие концепции и процедуры в Windows имеют свои корни в C++. Одна из таких концепций — *классы Windows*. В Windows каждое окно, управляющий элемент, диалоговое окно и прочее представляют собой окно, которое отличается от других окон *классом*, определяющим его вид и поведение. Класс Windows представляет собой описание типа окна, с которым работает Windows.

Существует ряд предопределенных классов Windows: кнопки, списки, диалоговые окна выбора файла и т.п., но это не мешает вам создавать собственные классы. В действительности при написании любого приложения необходимо создать, как минимум, один класс Windows. Класс можно рассматривать как шаблон, которому следует Windows при выводе вашего окна и при обработке сообщений для него.

Для хранения информации о классе Windows имеются две структуры: `WNDCLASS` и `WNDCLASSEX`. Первая структура более старая и, вероятно, вскоре выйдет из употребления, так что познакомимся со структурой `WNDCLASSEX`. Обе структуры очень похожи друг на друга, и если вас интересует первая из них, то вы можете обратиться к справочной системе Win32. Вот как определена структура `WNDCLASSEX` в заголовочном файле:

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

Ваша задача состоит в том, чтобы создать такую структуру и заполнить все ее поля.

```
WNDCLASSEX winclass;
```

Первое поле `cbSize` очень важно (даже Петцольд забыл об этом в своей книге *Programming Windows 95*). Оно представляет собой размер самой структуры `WNDCLASSEX`. Вас может удивить — зачем структуре знать, какого она размера? Хороший вопрос. Причина в том, что, если эта структура передается как указатель, получатель может всегда проверить первое поле структуры и выяснить, каков реальный размер передаваемого ему блока памяти. Можно считать это предосторожностью и небольшой подсказкой другим функци-

ям, чтобы им не пришлось вычислять размер класса в процессе работы[1]. Таким образом, вам необходимо лишь указать значение первого поля следующим образом:

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

Следующее поле содержит флаги стиля окна, которые описывают его основные свойства. Имеется множество этих флагов, так что мы опять ограничимся только некоторыми из них. Об остальных достаточно только знать, что с их помощью вы можете создать окно любого типа. В табл. 2.6 приведено рабочее подмножество этих флагов, которого достаточно для большинства случаев. Флаги могут быть объединены с помощью операции побитового ИЛИ.

Таблица 2.6. Некоторые флаги стиля для классов Windows

Флаг	*Описание*
CS_HREDRAW	Если при перемещении или коррекции размера изменяется ширина окна, требуется перерисовка всего окна
CS_VREDRAW	Если при перемещении или коррекции размера изменяется высота окна, требуется перерисовка всего окна
CS_OWNDC	Каждому окну данного класса выделяется свой контекст устройства (более подробно об этом будет сказано позже)
CS_DBLCLKS	При двойном щелчке мышью в тот момент, когда курсор находится в окне, процедуре окна посылается сообщение о двойном щелчке
CS_PARENTDC	Устанавливает область обрезки дочернего окна равной области обрезки родительского окна, так что дочернее окно может изображаться в родительском
CS_SAVEBITS	Сохраняет изображение в окне, так что вам не нужно перерисовывать его каждый раз, когда оно закрывается, перемещается и т.д. Однако для этого требуется много памяти, и обычно такая процедура медленнее, чем если бы вы перерисовывали окно сами
CS_NOCLOSE	Отключает команду Close в системном меню

Примечание. Наиболее часто употребляемые флаги выделены полужирным шрифтом.

Пока что из всего разнообразия флагов вам достаточно только нескольких, чтобы указать, что окно должно быть перерисовано при перемещении или изменении размера и что вам требуется статический *контекст устройства* (device context) наряду с возможностью обрабатывать двойные щелчки мышью.

Детальнее контекст устройства рассматривается в главе 3, "Профессиональное программирование в Windows", а пока лишь скажу, что это структура данных для вывода графической информации в окне. Следовательно, если вы хотите выводить в окне графическую информацию, вы должны запросить контекст устройства для данного окна. Если вы определите в классе Windows с помощью флага CS_OWNDC, что вам требуется собственный контекст устройства, вы сможете сохранить немного времени, поскольку не потребуется запрашивать контекст всякий раз при необходимости вывода. Таким образом, для наших целей вполне достаточно определить поле style.

[1] Еще одна причина, например, в том, что это поле позволяет в будущем легко расширить структуру, сохранив при этом обратную совместимость. Пояснение автора о том, что при таком подходе не требуется вычисление размера во время *работы*, не выдерживает критики, поскольку размер структуры вычисляется во время *компиляции*. — *Прим. ред.*

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

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

Именно в этом состоит суть цикла событий Windows и работы обработчика событий. Вы указываете функцию обратного вызова (само собой, с предопределенным прототипом) в описании класса. При осуществлении события Windows вызывает эту функцию, как показано на рис. 2.6. Более подробно этот вопрос рассматривается позже, а пока достаточно записать

```
winclass.lpfnWndProc = WinProc;
```

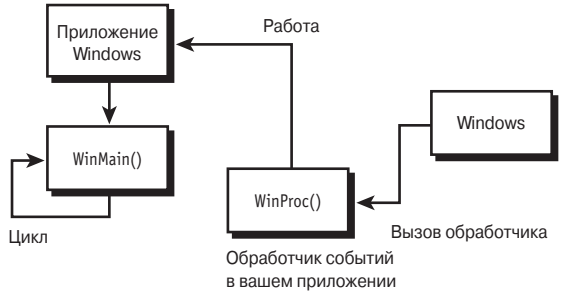


Рис. 2.6. Работа функции обратного вызова — обработчика событий

СОВЕТ

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

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

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

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

Теперь вы можете присвоить значение указателю и использовать его для вызова соответствующей функции:

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

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

Красиво, не правда ли?

Следующие два поля, cbClsExtra и cbWndExtra, были изначально созданы для хранения дополнительной информации времени исполнения. Однако большинство программистов не используют эти поля и устанавливают их равными 0.

```
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
```

Далее следует поле hInstance. Это та же величина, что и переданная в качестве параметра функции WinMain().

```
winclass.hInstance = hinstanse;
```

Остальные поля структуры относятся к различным графическим аспектам класса Windows, но перед их рассмотрением я бы хотел немного отвлечься, чтобы сказать пару слов о дескрипторах.

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

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

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

Этот код загружает стандартную пиктограмму приложения — невыразительно, зато очень просто. Если вас интересует функция LoadIcon(), то ниже приведен ее прототип, а в табл. 2.7 вы найдете различные варианты пиктограмм, загружаемые этой функцией.

```
HICON LoadIcon(
  HINSTANCE hInstance, // Экземпляр приложения
  LPCTSTR lpIconName); // Имя пиктограммы или
                       // идентификатор ее ресурса
```

Таблица 2.7. Идентификаторы пиктограмм, используемые функцией LoadIcon()

Значение	*Описание*
IDI_APPLICATION	Пиктограмма приложения по умолчанию
IDI_ASTERISK	Информация
IDI_EXCLAMATION	Восклицательный знак
IDI_HAND	Пиктограмма в виде руки
IDI_QUESTION	Вопросительный знак
IDI_WINLOGO	Эмблема Windows

В приведенном выше прототипе hInstance представляет собой экземпляр приложения, из ресурсов которого загружается пиктограмма (немного позже мы поговорим об этом подробнее), и для загрузки стандартных пиктограмм здесь используется значение NULL. А завершающаяся нулевым символом строка lpIconName содержит имя загружаемого ресурса пиктограммы. Если значение hInstance равно NULL, то lpIconName может принимать одно из приведенных в табл. 2.7 значений.

Итак, половину полей мы уже прошли. Приступим ко второй половине, которая начинается с поля hCursor. Оно похоже на поле hIcon в том плане, что также представляет собой дескриптор графического объекта. Однако hCursor отличается тем, что это дескриптор курсора, который изображается, когда указатель находится в клиентской области окна. Для получения дескриптора курсора используется функция LoadCursor(), которая так же, как и LoadIcon(), загружает изображение курсора из ресурсов приложения (о ресурсах речь идет несколько ниже).

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

Ниже приведен прототип функции LoadCursor(), а в табл. 2.8 приведены идентификаторы различных системных курсоров.

```
HCURSOR LoadCursor(
  HINSTANCE hInstance,   // Экземпляр приложения
  LPCTSTR lpCursorName); // Имя курсора или
                         // идентификатор его ресурса
```

Таблица 2.8. Идентификаторы курсоров для функции LoadCursor()

Значение	*Описание*
IDC_ARROW	Стандартная стрелка
IDC_APPSTARTING	Стандартная стрелка с маленькими песочными часами
IDC_CROSS	Крест
IDC_IBEAM	Текстовый I-курсор
IDC_NO	Перечеркнутый круг
IDC_SIZEALL	Курсор перемещения (четыре стрелки)
IDC_SIZENESW	Двойная стрелка (на северо-восток и юго-запад)
IDC_SIZENS	Двойная стрелка (на север и юг)
IDC_SIZENWSE	Двойная стрелка (на северо-запад и юго-восток)
IDC_SIZEWE	Двойная стрелка (на запад и восток)
IDC_UPARROW	Вертикальная стрелка
IDC_WAIT	Песочные часы

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

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

Функция GetStockObject() выдает дескриптор объекта из семейства кистей, перьев, палитр или шрифтов Windows. Она получает один параметр, указывающий, какой именно ресурс следует загрузить. В табл. 2.9 приведен список возможных объектов (только кисти и перья).

Таблица 2.9. Идентификаторы объектов GetStockObject()

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

Очередное поле структуры WNDCLASS — lpszMenuName. Это завершающаяся нулевым символом строка с именем ресурса меню, которое следует загрузить и присоединить к окну. И на эту тему мы поговорим позже, в главе 3, "Профессиональное программирование в Windows", а пока просто установим значение этого поля равным NULL.

```
winclass.lpszMenuName = NULL;
```

Как уже упоминалось, каждый класс Windows представляет отдельный тип окна, которое может создать ваше приложение. Windows требуется каким-то образом различать разные классы окон, для чего и служит поле lpszClassName. Это завершающаяся нулевым символом строка с текстовым идентификатором данного класса. Лично я предпочитаю использовать в качестве имен строки типа "WINCLASS1", "WINCLASS2" и т.д. Впрочем, это дело вкуса.

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

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

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

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

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

```
WNDCLASSEX winclass;
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_VREDRAW | CS_HREDRAW |
         CS_OWNDC  | CS_DBLCKLICKS;
winclass.lpfnWndProc = WinProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstanse;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

```
winclass.lpszMenuName  = NULL;
winclass.lpszClassName = "WINCLASS1";
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

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

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

Регистрация класса Windows

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

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

ВНИМАНИЕ

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

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

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

Создание окна

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

Вот как выглядит прототип функции CreateWindowEx():

```
HWND CreateWindowEx(
  DWORD dwExStyle,      // Дополнительный стиль окна
  LPCTSTR lpClassName,  // Указатель на имя
                        // зарегистрированного класса
  LPCTSTR lpWindowName, // Указатель на имя окна
  DWORD dwStyle,        // Стиль окна
```

```
int x,                  // Горизонтальная позиция окна
int y,                  // Вертикальная позиция окна
int nWidth,             // Ширина окна
int nHeight,            // Высота окна
HWND hWndParent,        // Дескриптор родительского окна
HWND hMenu,             // Дескриптор меню или
                        // идентификатор дочернего окна
HINSTANCE hInstance,    // Дескриптор экземпляра приложения
LPVOID lpParam);        // Указатель на данные создания окна
```

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

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

- dwExStyle. Флаг дополнительных стилей окна; в большинстве случаев просто равен NULL. Если вас интересует, какие именно значения может принимать данный параметр, обратитесь к справочной системе Win32 SDK. Единственный из этих параметров, время от времени использующийся мною, — WS_EX_TOPMOST, который "заставляет" окно находиться поверх других.
- lpClassName. Имя класса, на основе которого создается окно.
- lpWindowName. Завершающаяся нулевым символом строка с заголовком окна, например "Мое первое окно".
- dwStyle. Флаги, описывающие, как должно выглядеть и вести себя создаваемое окно. Этот параметр очень важен! В табл. 2.10 приведены некоторые наиболее часто используемые флаги (которые, как обычно, можно объединять с помощью операции побитового ИЛИ).
- x, y. Позиция верхнего левого угла окна в пикселях. Если положение окна при создании не принципиально, воспользуйтесь значением CW_USEDEFAULT, и Windows разместит окно самостоятельно.
- nWidth, nHeight. Ширина и высота окна в пикселях. Если размер окна при создании не принципиален, воспользуйтесь значением CW_USEDEFAULT, и Windows выберет размеры окна самостоятельно.
- hWndParent. Дескриптор родительского окна, если таковое имеется. Значение NULL указывает, что у создаваемого окна нет родительского окна и в качестве такового должен считаться десктоп.
- hMenu. Дескриптор меню, присоединенного к окну. О нем речь идет в следующей главе, а пока будем использовать значение NULL.
- hInstance. Экземпляр приложения. Здесь используется значение параметра hinstance функции WinMain().
- lpParam. Пока что мы просто устанавливаем это значение равным NULL.

Таблица 2.10. Значения стилей, использующиеся в параметре dwStyle

Значение	*Описание*
WS_POPUP	Всплывающее окно
WS_OVERLAPPED	Перекрывающееся окно с полосой заголовка и рамкой. То же, что и стиль WS_TILED

Окончание табл. 2.10

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

Примечание. Наиболее часто употребляющиеся значения выделены полужирным шрифтом.

Вот как создается обычное перекрывающееся окно со стандартными управляющими элементами в позиции (0, 0) размером 400×400 пикселей:

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

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

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

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

Обработчик событий

Для того чтобы освежить память и вспомнить о том, для чего нужен обработчик событий и что такое функция обратного вызова, вернитесь к рис. 2.6.

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

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

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

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

Рассмотрим теперь прототип WinProc.

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

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

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

А теперь познакомимся с параметрами этой функции.

- hwnd. Дескриптор окна. Этот параметр важен в том случае, когда открыто несколько окон одного и того же класса. Тогда hwnd позволяет определить, от какого именно окна поступило сообщение (рис. 2.7).
- msg. Идентификатор сообщения, которое должно быть обработано WinProc.
- wparam и lparam. Эти величины являются параметрами обрабатываемого сообщения, несущими дополнительную информацию о нем.

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

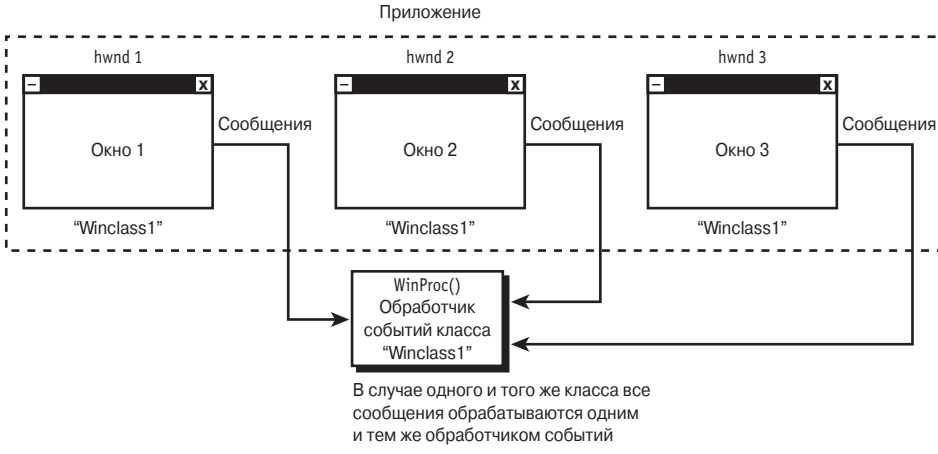


Рис. 2.7. Работа нескольких окон одного класса

Большинство программистов при написании обработчика событий используют конструкцию switch(msg), в которой на основании значения msg принимается решение об использовании дополнительных параметров wparam и/или lparam. В табл. 2.11 приведены некоторые из возможных идентификаторов сообщений.

Таблица 2.11. Краткий список идентификаторов сообщений

Значение	*Описание*
WM_ACTIVATE	Посылается, когда окно активизируется или получает фокус ввода
WM_CLOSE	Посылается, когда окно закрывается
WM_CREATE	Посылается при создании окна
WM_DESTROY	Посылается, когда окно должно быть уничтожено
WM_MOVE	Посылается при перемещении окна
WM_MOUSEMOVE	Посылается при перемещении мыши
WM_KEYUP	Посылается при отпускании клавиши
WM_KEYDOWN	Посылается при нажатии клавиши
WM_TIMER	Посылается при наступлении события таймера
WM_USER	Позволяет вам посылать собственные сообщения
WM_PAINT	Посылается при необходимости перерисовки окна
WM_QUIT	Посылается при завершении работы приложения
WM_SIZE	Посылается при изменении размеров окна

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

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

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

Вот простейшая функция WinProc, обрабатывающая описанные сообщения:

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

  // Какое сообщение получено?
  switch(msg)
  {
    case WM_CREAYE:
    {              // Выполнение инициализации
      return(0); // Успешное выполнение
    } break;

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

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

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

    default: break;

  } // switch

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

} // WindowProc
```

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

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

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

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

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

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

Сейчас, пока не рассматривалась работа с GDI, вам незачем беспокоиться о всех полях этой структуры — вы познакомитесь с ними позже. Здесь же стоит познакомиться поближе только с полем rcPaint, которое представляет собой структуру RECT, указывающую минимальный прямоугольник, требующий перерисовки. Взгляните на рис. 2.8, поясняющий это. Windows пытается обойтись минимальным количеством действий, поэтому при необходимости перерисовки окна старается вычислить минимальный прямоугольник, перерисовать который было бы достаточно для восстановления содержимого всего окна. Если вас интересует, что конкретно представляет собой структура RECT, то это не более чем четыре угла прямоугольника.

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

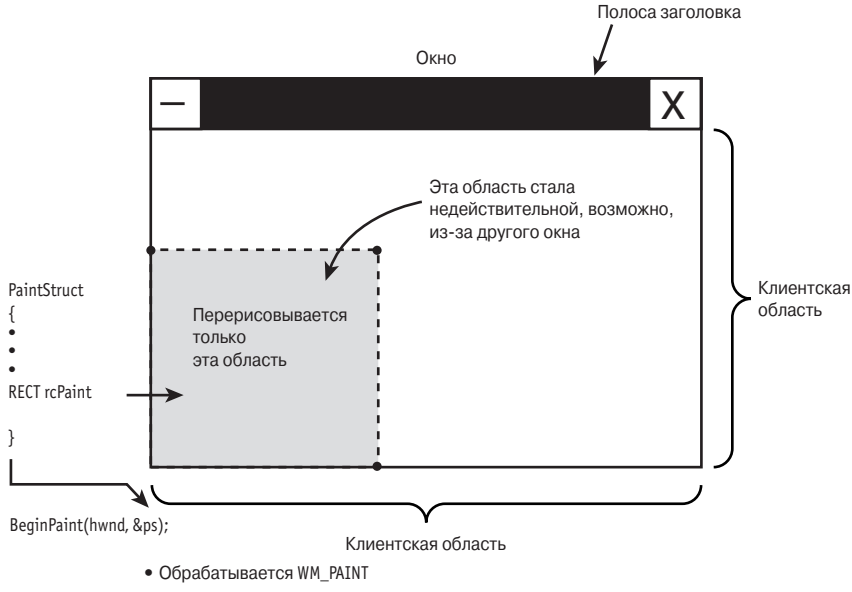


Рис. 2.8. Перерисовывается только недействительная область

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

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

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

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

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

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

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

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

Главный цикл событий

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

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

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

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

```
BOOL GetMessage(
  LPMSG lpMsg,          // Адрес структуры с сообщением
  HWND  hWnd,           // Дескриптор окна
  UINT  wMsgFilterMin,  // Первое сообщение
  UINT  wMsgFilterMax); // Последнее сообщение
```

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

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

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

Итак, функция GetMessage() получает очередное сообщение из очереди событий. А что затем? Следующей вызывается функция TranslateMessage(), которая представляет собой транслятор виртуальных "быстрых клавиш". Ваше дело — вызвать эту функцию, остальное — не ваша забота. Последняя вызываемая функция — DispatchMessage(), которая, собственно, и выполняет всю работу по обработке сообщений. После того как сообщение получено с помощью функции GetMessage() и преобразовано функцией TranslateMessage(), функция DispatchMessage() вызывает для обработки функцию WinProc(), передавая ей всю необходимую информацию, которая находится в структуре MSG. На рис. 2.9 показан весь описанный процесс.

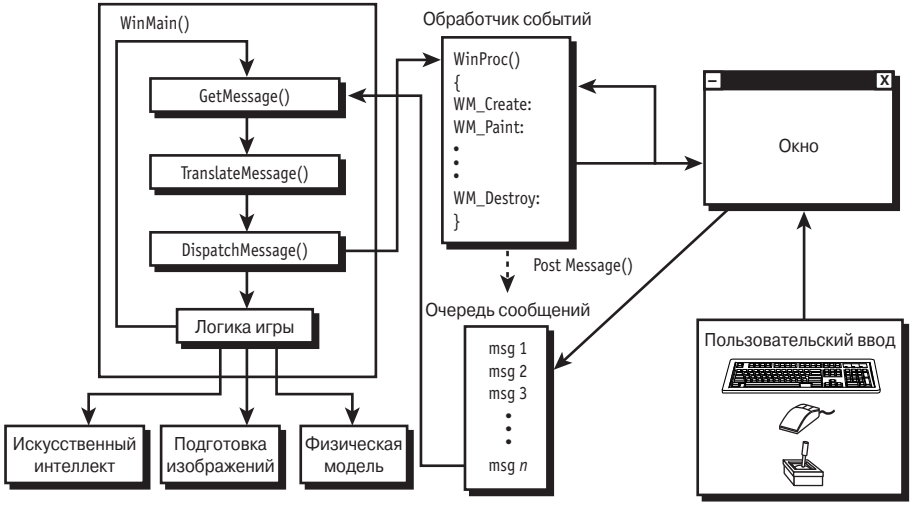


Рис. 2.9. Механизм обработки сообщений

Вот и все; если вы разобрались в этом, можете считать себя программистом в Windows. Остальное — не более чем небольшие детали. Посмотрите с этой точки зрения на листинг 2.3, где представлена завершенная Windows-программа, которая просто создает одно окно и ждет, пока вы его не закроете.

Листинг 2.3. Базовая Windows-программа

```
// DEMO2_3.CPP - Базовая Windows-программа

// Включаемые файлы /////////////////////////////////////
#define WIN32_LEAN_AND_MEAN  // He MFC :-)

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

// Определения //////////////////////////////////////////

#define WINDOW_CLASS_NAME "WINCLASS1"

// Глобальные переменные ////////////////////////////////


// Функции //////////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam)
{
  // Главный обработчик сообщений в системе
  PAINTSTRUCT  ps;     // Используется в WM_PAINT
  HDC          hdc;    // Дескриптор контекста устройства
  // Какое сообщение получено
  switch(msg)
```

```
  {
    case WM_CREATE:
      {
        // Действия по инициализации
        return(0);  // Успешное завершение
      } break;
    case WM_PAINT:
      {
        // Объявляем окно действительным
        hdc = BeginPaint(hwnd,&ps);
        // Здесь выполняются все действия
        // по выводу изображения
        EndPaint(hwnd,&ps);
        return(0);  // Успешное завершение
      } break;
    case WM_DESTROY:
      {
        // Завершение работы приложения
        PostQuitMessage(0);

        // Успешное завершение
        return(0);
      } break;
    default:break;
  } // switch
  // Обработка остальных сообщений
  return (DefWindowProc(hwnd, msg, wparam, lparam));
} // WinProc

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


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

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

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

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

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

//////////////////////////////////////////////////////////
```

Для компиляции программы создайте проект Win32 .EXE и добавьте к нему DEMO2_3.CPP (вы можете также просто запустить скомпилированную программу DEMO2_3.EXE, находящуюся на прилагаемом компакт-диске). На рис. 2.10 показана работа данной программы.

Рис. 2.10. Работа программы DEMO2_3.EXE

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

Цикл сообщений для работы в реальном времени

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

```
BOOL PeekMessage(
  LPMSG lpMsg,          // Адрес структуры с сообщением
  HWND  hWnd,           // Дескриптор окна
  UINT  wMsgFilterMin,  // Первое сообщение
  UINT  wMsgFilterMax,  // Последнее сообщение
  UINT wRemovwMsg);   // Флаг удаления сообщения
```

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

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

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

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

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

    TranslateMessage(&msg);

    DispatchMessage(&msg);
  } // if

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

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

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

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

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

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

Открытие нескольких окон

Прежде чем завершить данную главу, я хочу вкратце затронуть еще одну тему: как открыть несколько окон одновременно. На самом деле это тривиальная задача, ответ которой вы уже знаете. Для этого требуется лишь вызвать необходимое количество раз функцию CreateWindowEx(). Однако здесь есть некоторые тонкости.

Вспомним, что, когда вы создаете окно, его свойства определяются классом. Этот класс, помимо прочего, определяет обработчик событий для данного класса. Следовательно, вы можете создать сколько угодно окон данного класса, но *все* сообщения для них будут обрабатываться одним и тем же обработчиком, указанным в поле lpfnWndProc структуры WNDCLASSEX. На рис. 2.11 представлена диаграмма потока сообщений в данной ситуации.

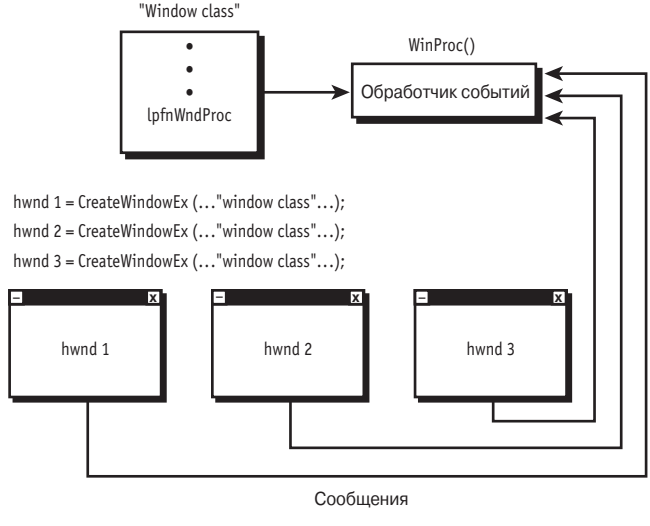


Рис. 2.11. Поток сообщений при наличии нескольких окон одного класса

Это может быть именно тем, что вам нужно, но если вы хотите, чтобы у каждого окна была своя процедура WinProc, следует создать несколько классов Windows с тем, чтобы классы создаваемых вами окон были различны. В такой ситуации, поскольку классы

окон различны, сообщения для разных окон будут пересылаться разным обработчикам событий, как показано на рис. 2.12.

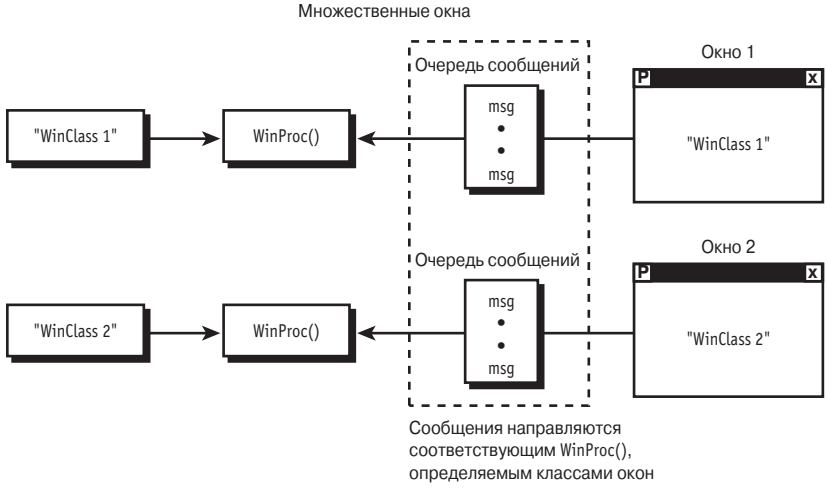


Рис. 2.12. Сообщения в системе с несколькими окнами различных классов

Вот код, создающий два окна на основе одного и того же класса:

```
// Создаем первое окно
if (!(hwnd = CreateWindowEx(
      NULL,
      WINDOW_CLASS_NAME,    // Класс окна
      "Window 1 Based on WINCLASS1",
      WS_OVERLAPPEDWINDOW | WS_VISIBLE,
      0,0,
      400,400,
      NULL,
      NULL,
      hinstance,
      NULL)))
return(0);

// Создаем второе окно
if (!(hwnd = CreateWindowEx(
      NULL,
      WINDOW_CLASS_NAME,    // Класс окна
      "Window 2 Based on WINCLASS1",
      WS_OVERLAPPEDWINDOW | WS_VISIBLE,
      100,100,
      400,400,
      NULL,
      NULL,
      hinstance,
      NULL)))
return(0);
```

Разумеется, вы можете хранить дескрипторы окон в разных переменных, а не в одной, как в данном фрагменте. Главное — уловить идею. В качестве примера приложения, открывающего одновременно два окна, рассмотрите приложение DEMO2_5.CPP, находящееся на прилагаемом компакт-диске. При запуске DEMO2_5.EXE вы увидите два окна, как показано на рис. 2.13. Обратите внимание, что при закрытии любого из окон приложение завершает свою работу. Попробуйте изменить приложение таким образом, чтобы оно не завершалось до тех пор, пока не будут закрыты оба окна. (Подсказка: создайте два класса Windows и не посылайте сообщение WM_QUIT, пока не будут закрыты оба окна.)

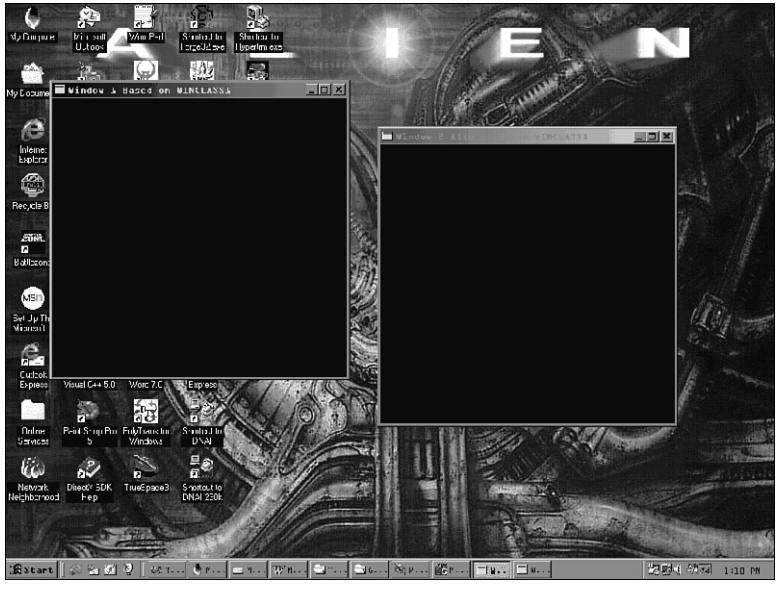


Рис. 2.13. Программа DEMO2_5.EXE с двумя окнами

Резюме

Теперь вы знаете все необходимое для того, чтобы приступить к рассмотрению более сложных аспектов программирования в Windows. Вы уже знакомы с архитектурой Windows и многозадачностью, знаете, как создать класс Windows, зарегистрировать его, создать окно, написать цикл событий и обработчики, и многое другое. Теперь вы вправе немного передохнуть и с полной силой взяться за новый материал.

В следующей главе вы встретитесь с такими вещами, как использование ресурсов, создание меню, работа с диалоговыми окнами и получение информации.

ГЛАВА 3

Профессиональное программирование в Windows

Не надо быть семи пядей во лбу, чтобы сообразить, что программирование в Windows — огромная тема. Но самое интересное в том, что вам и не нужно знать все в мельчайших подробностях. В этой главе рассматриваются только некоторые наиболее важные вопросы, знание которых необходимо для создания завершенного приложения Windows.

- Использование ресурсов, таких, как пиктограммы, курсоры, звуки
- Использование меню
- Базовый GDI и видеосистема
- Устройства ввода
- Отправка сообщений

Использование ресурсов

При создании Windows одним из важных вопросов проектирования была проблема хранения дополнительной информации (помимо кода программы) в файле приложения Windows. Разработчики обосновывали это тем, что данные для программы должны располагаться вместе с кодом, в одном .EXE-файле, поскольку при этом облегчается распространение приложения, представляющего собой единственный файл, при таком хранении дополнительные данные невозможно потерять, а также труднее исказить или украсть.

Для упрощения этой методики Windows поддерживает *ресурсы*. Это просто фрагменты данных, объединенные с кодом вашей программы в одном файле, которые могут быть загружены позже, в процессе выполнения самой программы. Данная концепция показана на рис. 3.1.

Рис. 3.1. Взаимосвязь ресурсов и приложения Windows

О каких именно типах ресурсов идет речь? В действительности нет никаких ограничений на типы данных, которые можно скомпоновать с вашей программой, поскольку Windows поддерживает типы ресурсов, *определенные пользователем*. Однако существует целый ряд предопределенных типов, которые должны удовлетворять большинство ваших потребностей.

- *Пиктограмма*. Изображения малого размера, использующиеся в самых разных ситуациях, например представляющие приложение в каталоге. Отдельные пиктограммы находятся в файлах с расширением .ICO.
- *Курсор*. Изображение малого размера для представления указателя мыши. Windows позволяет управлять внешним видом курсора, например изменять его при переходе от окна к окну. Курсоры располагаются в файлах с расширением .CUR.
- *Строка*. Использование ресурса этого типа не так очевидно. Как правило, приходится слышать возражение "я размещаю строки непосредственно в программе; на худой конец — в файле данных". Я могу вас понять, и тем не менее Windows позволяет размещать таблицу строк в файле приложения в качестве ресурсов и обращаться к ним посредством идентификатора ресурса.
- *Звук*. Многие программы Windows используют звуковые .WAV-файлы. Эти звуковые данные также могут быть добавлены к ресурсам вашего приложения. Между прочим, неплохой метод спрятать звуковые эффекты от похищения неопытными пользователями.
- *Изображение*. Стандартные растровые изображения, представляющие собой прямоугольную матрицу пикселей в монохромном или 4-, 8-, 16-, 24- или 32-битовом цветном формате. Это широко распространенные объекты в графических операционных системах, таких, как Windows. Такие изображения хранятся в файлах с расширением .BMP.
- *Диалог*. Диалоговое окно представляет собой широко распространенный объект в Windows, поэтому было бы разумнее сделать его ресурсом, а не хранить как внешний файл. Таким образом, диалоговые окна могут создаваться как "на лету" кодом

вашего приложения, так и разрабатываться в специальном редакторе, а затем храниться в качестве ресурса.

- *Метафайл.* Более передовая технология. Они позволяют записать последовательность графических операций в файл, а затем воспроизвести этот файл.

Теперь, когда вы уловили основную идею и познакомились с возможными типами ресурсов, возникает вопрос о том, как же объединить ресурсы в файле приложения? Для этого служит программа, называющаяся *компилятором ресурсов*. Она получает на вход текстовый файл ресурсов (с расширением .RC), представляющий собой описание всех ресурсов, которые требуется скомпилировать в единый файл данных, после чего загружает все описанные ресурсы и компонует их в один большой файл данных с расширением .RES.

Этот файл содержит бинарные данные, представляющие собой пиктограммы, курсоры, изображения и т.д., описанные в .RC-файле. Затем .RES-файл вместе со всеми прочими файлами (.CPP, .H, .LIB, .OBJ и т.д.) участвует в компиляции и компоновке единого .EXE-файла приложения. На рис. 3.2 показана схема данного процесса.

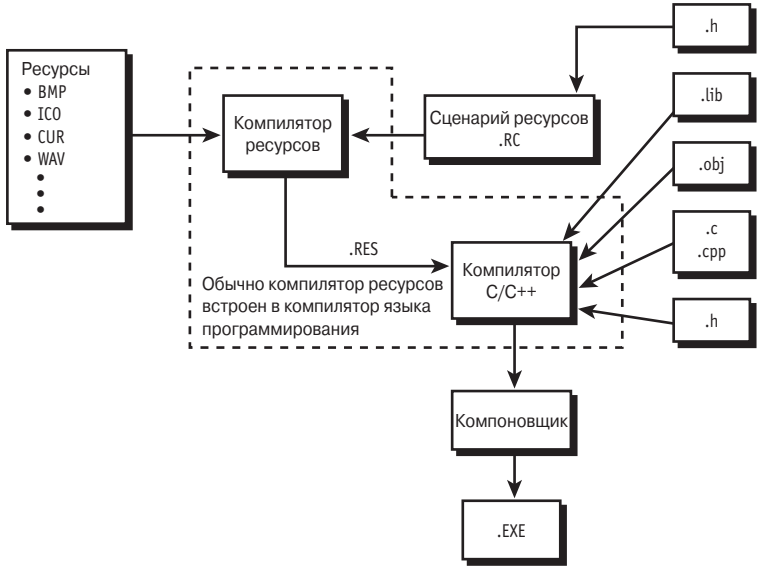


Рис. 3.2. Компиляция и компоновка приложения с ресурсами

Размещение и работа с ресурсами

Раньше для компиляции и размещения ресурсов использовался внешний компилятор ресурсов типа RC.EXE. Однако в последнее время его функции берут на себя интегрированные среды программирования. Следовательно, если вы хотите добавить ресурсы в вашу программу, то можете просто добавить их в проект, выбрав пункт меню File⇨New в интегрированной среде, а затем указав, какой именно тип ресурса следует добавить.

Вкратце рассмотрим, как осуществляется работа с ресурсами. Вы можете добавить любое количество объектов разных типов в вашу программу, и они будут размещены в качестве ресурсов в вашем .EXE-файле наряду с кодом программы. Затем во время выполнения программы вы можете получить доступ к этой базе данных ресурсов и загрузить необходимые данные из файла программы, а не из отдельного файла на диске. Для создания файла ресурсов вы должны организовать текстовый файл описания ресурсов с

расширением .RC. Вместе с доступом к самим файлам ресурсов он передается компилятору, в результате чего генерируется бинарный .RES- файл. Этот файл затем компонуется вместе с другими объектами программы в один .EXE-файл. Все очень просто.

Познакомившись с принципами работы ресурсов, мы рассмотрим объекты разных типов и узнаем, как они создаются и как загружаются при работе программы. Все возможные типы ресурсов рассматриваться не будут, но приведенного в книге материала должно быть достаточно, чтобы при необходимости вы смогли разобраться с ресурсами других типов самостоятельно.

Использование ресурсов пиктограмм

При работе с ресурсами вы должны создать два файла — .RC-файл и, возможно, .H-файл (если хотите использовать символьные идентификаторы в .RC-файле). Ну и, разумеется, вам необходим .RES-файл, но об этом позаботится интегрированная среда разработки.

В качестве примера создания ресурса пиктограммы рассмотрим, как изменить пиктограмму, которую приложение использует в панели задач и системном меню окна. Вспомните, что ранее мы определяли эти пиктограммы при создании класса Windows следующим образом:

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

Этими строками кода загружались пиктограммы по умолчанию (как обычная пиктограмма, так и уменьшенная). Однако можно загрузить любую пиктограмму, какую вы только пожелаете, воспользовавшись возможностью размещения пиктограммы как ресурса.

Прежде всего нам нужна пиктограмма, с которой мы будем работать. С этой целью я создал пиктограмму для всех приложений в данной книге (ее имя — T3DX.ICO, а как она выглядит, можно увидеть на рис. 3.3). Я создал эту пиктограмму с помощью VC++ 6.0 Image Editor, который показан на рис. 3.4. Однако вы можете использовать для создания пиктограмм, курсоров, изображений и прочего любой другой инструментарий, приспособленный для этого.

Рис. 3.3. Пиктограмма T3DX.ICO

T3DX.ICO представляет собой 16-цветную пиктограмму размером 32×32 пикселя. Размер пиктограмм может находиться в диапазоне от 16×16 до 64×64 пикселя, с количеством цветов до 256. Однако большинство пиктограмм 16-цветные размером 32×32 пикселя.

После создания пиктограммы ее следует поместить в файл ресурса. Конечно, интегрированная среда разработки способна взять решение этой задачи на себя, но, согласитесь, для обучения гораздо полезнее сделать это вручную.

Файл .RC содержит все определения ресурсов; это означает, что в вашей программе их может быть множество.

НА ЗАМЕТКУ

Перед тем как приступить к написанию кода, я хотел бы отметить одну важную вещь. Windows использует для обращения к ресурсам либо текстовую ASCII-строку, либо целочисленный идентификатор. В большинстве случаев в .RC-файле можно использовать оба варианта. Однако некоторые ресурсы позволяют использовать хотя и любой способ, но только один. При использовании идентификаторов в проект должен быть включен дополнительный .H-файл, содержащий символьные перекрестные ссылки.

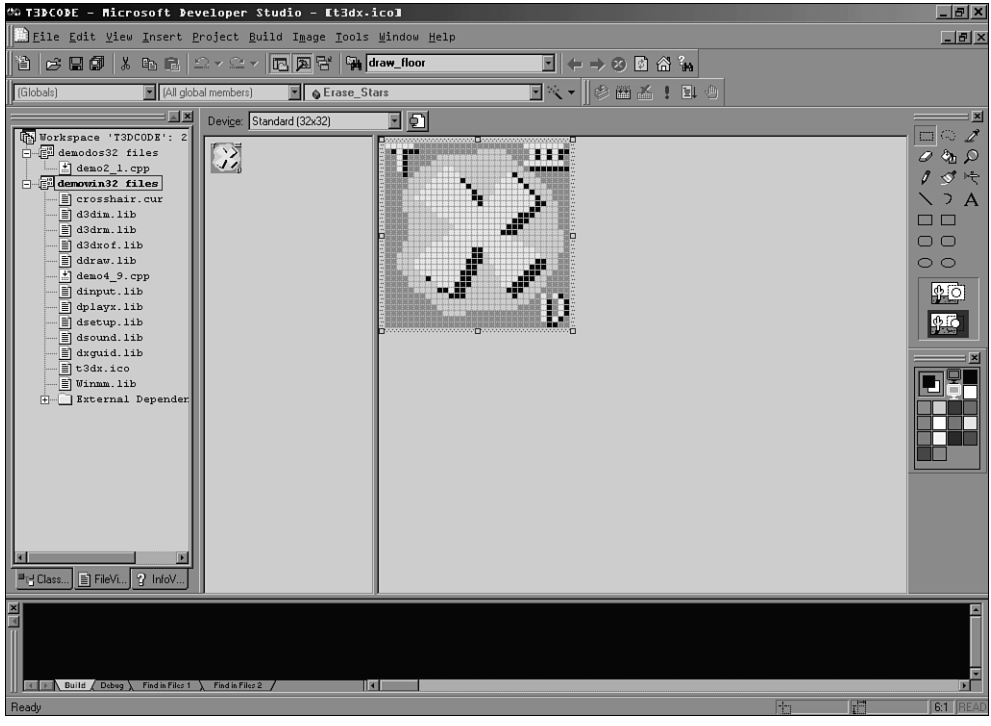


Рис. 3.4. VC++ 6.0 Image Editor

Вот как мы можем определить ресурс ICON в .RC-файле.

Метод 1 — с использованием строкового имени:

```
icon_name ICON FILENAME.ICO
```

Например:

```
windowicon ICON star.ico
MyCoolIcon ICON cool.ico
```

Метод 2 — с использованием целочисленного идентификатора:

```
icon_id ICON FILENAME.ICO
```

Например:

```
windowicon ICON star.ico
124      ICON ship.ico
```

Обратите внимание на отсутствие кавычек при использовании первого метода. Это создает определенные проблемы, например код в .RC-файле может выглядеть совершенно одинаково для разных методов. Однако в первом случае компилятор интерпретирует первое слово как "windowicon", а во втором — как символьное имя windowicon. Что именно происходит, определяется дополнительным файлом, который вы включаете в ваш .RC-файл (и .CPP-файл вашего приложения), и в котором определяются все символьные константы. Предположим, компилятору нужно разобрать строку

```
windowicon ICON star.ico
```

При этом он просматривает все символьные имена, определенные во включаемых заголовочных файлах. Если такое имя имеется, то компилятор ресурсов обращается к ре-

сурсу по целочисленному идентификатору, определяемому данным именем. В противном случае компилятор считает, что это строка, и обращается к пиктограмме с использованием имени "windowicon".

Таким образом, если вы хотите использовать символьные идентификаторы в .RC-файле, вам необходим соответствующий .H-файл для разрешения символьных имен. Этот файл включается в .RC-файл при помощи стандартной директивы препроцессора C/C++ #include.

Предположим, например, что вы хотите определить три пиктограммы с использованием символьных имен в файле, который мы назовем RESOURCE.RC. В таком случае нам потребуется заголовочный файл RESOURCE.H. Приведем содержимое этих файлов.

Файл RESOURCE.H:

```
#define ID_ICON1   100 // Это произвольные числа
#define ID_ICON2   101
#define ID_ICON3   102
```

Файл RESOURCE.RC:

```
#include "RESOURCE.H"

// Определения пиктограмм (Обратите внимание на
// возможность использования комментариев в стиле C++)

ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

Теперь вы можете добавить файл RESOURCE.RC в ваш проект и внести строку

```
#include "RESOURCE.H"
```

в файл приложения. Конечно, все необходимые .ICO-файлы также должны находиться в рабочем каталоге вашего проекта с тем, чтобы компилятор ресурсов мог их найти.

Если же вы не определите символьные имена и не включите файл RESOURCE.H в файл описания ресурсов, символьные имена ID_ICON1, ID_ICON2 и ID_ICON3 будут восприняты как строки и обращаться к данным пиктограммам в программе вы будете должны по именам "ID_ICON1", "ID_ICON2" и "ID_ICON3" соответственно.

Теперь, основательно разобравшись в тонкостях внесения пиктограмм в файл ресурсов, попробуем загрузить пиктограмму в нашем приложении.

Для того чтобы загрузить пиктограмму по строковому имени, поступайте следующим образом.

В .RC-файле определите пиктограмму:

```
your_icon_name ICON filename.ico
```

А в коде программы загрузите ее:

```
// Обратите внимание на использование hinstance вместо NULL
winclass.hIcon  = LoadIcon(hinstance, "your_icon_name");
winclass.hIconSm = LoadIcon(hinstance, "your_icon_name");
```

Для загрузки с использованием представленных символьными именами целочисленных идентификаторов вы должны включить при помощи директивы #include заголовочный файл как в .RC-, так и в .CPP-файл (в приведенных ниже фрагментах эта директива не показана).

Содержимое .H-файла:

```
#define ID_ICON1   100 // Это произвольные числа
#define ID_ICON2   101
#define ID_ICON3   102
```

Содержимое .RC-файла:

```
// Определения пиктограмм (Обратите внимание на
// возможность использования комментариев в стиле C++)

ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

Загрузка пиктограмм в программе:

```
// Обратите внимание на использование hinstance вместо NULL
// Макрос MAKEINTRESOURCE используется для корректного
// обращения к ресурсу при помощи символьной константы
winclass.hIcon  = LoadIcon(hinstance,
                  MAKEINTRESOURCE(ID_ICON1));
winclass.hIconSm = LoadIcon(hinstance,
                  MAKEINTRESOURCE(ID_ICON1));
```

Обратите внимание на использование макроса MAKEINTRESOURCE, который преобразует целое число в указатель на строку. Как именно он это делает — неважно; наша задача — просто использовать необходимые константы.

Использование ресурсов курсоров

Ресурсы курсоров очень похожи на ресурсы пиктограмм. Файлы курсоров (с расширением .CUR) представляют собой небольшие растровые изображения и могут быть легко созданы в большинстве интегрированных сред разработки или в графических редакторах. Обычно курсор представляет собой 16-цветное изображение размером 32×32 пикселя, но он может достигать размера 64×64 пикселя с 256 цветами и даже быть анимированным!

Предположим, что у нас уже имеется созданный курсор и теперь нужно добавить его в .RC-файл. Это делается так же, как и в случае пиктограммы.

Метод 1 — с использованием строкового имени:

```
cursor_name CURSOR FILENAME.CUR
```

Например:

```
windowcursor  CURSOR  star.cur
MyCoolCursor  CURSOR  cool.cur
```

Метод 2 — с использованием целочисленного идентификатора:

```
cursor_id CURSOR FILENAME.CUR
```

Например:

```
windowcursor  CURSOR  star.cur
124       CURSOR  ship.cur
```

Разумеется, при использовании символьных имен вы должны создать .H-файл с определениями символьных имен.

Файл RESOURCE.H:

```
#define ID_CURSOR_CROSSHAIR  200 // Это произвольные числа
#define ID_CURSOR_GREENARROW
```

Файл RESOURCE.RC:

```
#include "RESOURCE.H"

// Определения курсоров (Обратите внимание на
// возможность использования комментариев в стиле C++)

ID_CURSOR_CROSSHAIR  CURSOR  crosshair.cur
ID_CURSOR_GREENARROW CURSOR  greenarrow.cur
```

Нет никаких причин, по которым все файлы должны "толпиться" в одном рабочем каталоге. Их можно совершенно спокойно распределить по разным каталогам, чтобы не засорять рабочий, и соответственно указать в файле ресурсов полные имена файлов:

```
ID_CURSOR_GREENARROW CURSOR C:\CURSOR\greenarrow.cur
```

СЕКРЕТ

Я создал несколько файлов с курсорами для этой главы (вы найдете их на прилагаемом компакт-диске). Посмотрите на них с помощью соответствующих инструментов в вашей интегрированной среде, а потом просто откройте этот каталог в Windows и посмотрите на изображения рядом с именами соответствующих .CUR-файлов.

Теперь, когда вы познакомились со способами добавления курсоров в файл ресурсов, рассмотрите код, загружающий курсор, и сравните его с кодом загрузки пиктограммы.

Для того чтобы загрузить курсор по строковому имени, поступайте следующим образом.

В .RC-файле определите курсор:

```
CrossHair CURSOR crosshair.cur
```

А в коде программы загрузите его:

```
// Обратите внимание на использование hinstance вместо NULL
winclass.hCursor = LoadCursor(hinstance, "CrossHair");
```

Для загрузки с использованием представленных символьными именами целочисленных идентификаторов вы должны включить при помощи директивы #include заголовочный файл как в .RC-, так и в .CPP-файл (в приведенных ниже фрагментах эта директива не показана).

Содержимое .H-файла:

```
#define ID_CROSSHAIR  200
```

Содержимое .RC-файла:

```
ID_CROSSHAIR CURSOR crosshair.cur
```

Загрузка курсора в программе:

```
// Обратите внимание на использование hinstance вместо NULL
winclass.hCursor = LoadCursor(hinstance,
                   MAKEINTRESOURCE(ID_CROSSHAIR));
```

Здесь мы опять используем макрос MAKEINTRESOURCE для преобразования целого числа в указатель на строку, который требуется Windows.

Остается уточнить один маленький вопрос. Итак, мы можем загрузить пиктограммы и курсор и использовать их при определении класса. Но нельзя ли управлять ими на уровне окна? Например, вы создаете два окна одного класса, но хотите, чтобы курсор в этих окнах был разным. Для этого можно использовать функцию

```
HCURSOR SetCursor(HCURSOR hCursor);
```

Параметр hCursor представляет собой дескриптор курсора, полученный при помощи функции LoadCursor(). Единственная проблема при этом заключается в том, что функция SetCursor() не слишком интеллектуальна, так что ваше приложение должно отслеживать перемещение курсора мыши и изменять его при переходе из окна в окно. Вот пример установки нового курсора:

```
// Загрузка курсора (возможно, при обработке WM_CREATE)
HCURSOR hcrosshair = LoadCursor(hinstance, "CrossHair");

...

// Позже в программе для изменения вида курсора
SetCursor(hcrosshair);
```

Для того чтобы познакомиться с примером приложения с загрузкой пиктограммы окна и курсора мыши из ресурсов, обратитесь к программе DEMO3_1.CPP на прилагаемом компакт-диске. Вот фрагменты исходного текста программы, связанные с загрузкой пиктограммы и курсора.

```
#include "DEMO3_1RES.H" // включение заголовка ресурсов
.
.
winclass.hIcon   = LoadIcon(hinstance,
                 MAKEINTRESOURCE(ICON_T3DX));
winclass.hCursor = LoadCursor(hinstance,
                 MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hIconSm = LoadIcon(hinstance,
                 MAKEINTRESOURCE(ICON_T3DX));
```

Кроме того, программа использует сценарий ресурсов с именем DEMO3_1.RC и заголовок ресурсов DEMO3_1RES.H.

Содержимое DEMO3_1.RC:

```
#include "DEMO3_1RES.H"

// Обратите внимание на использование разных
// типов ресурсов в одном файле
ICON_T3DX     ICON  t3dx.ico
CURSOR_CROSSHAIR CURSOR crosshair.cur
```

Содержимое DEMO3_1RES.H:

```
#define ICON_T3DX          100
#define CURSOR_CROSSHAIR       200
```

Для создания приложения вам требуются следующие файлы:

DEMO3_1.CPP	Главный C/C++ файл
DEMO3_1RES.H	Заголовок с определениями символьных имен
DEMO3_1.RC	Сценарий ресурсов
T3DX.ICO	Изображение пиктограммы
CROSSHAIR.CUR	Изображение курсора

Все эти файлы должны находиться в одном и том же каталоге, что и ваш проект. В противном случае компилятор и компоновщик будут иметь проблемы при поиске файлов. После того как вы скомпилируете и запустите программу DEMO3_1.EXE, вы увидите на экране окно, показанное на рис. 3.5. Согласитесь, неплохой результат.

Рис. 3.5. Программа DEMO3_1.EXE с нестандартными пиктограммой и курсором

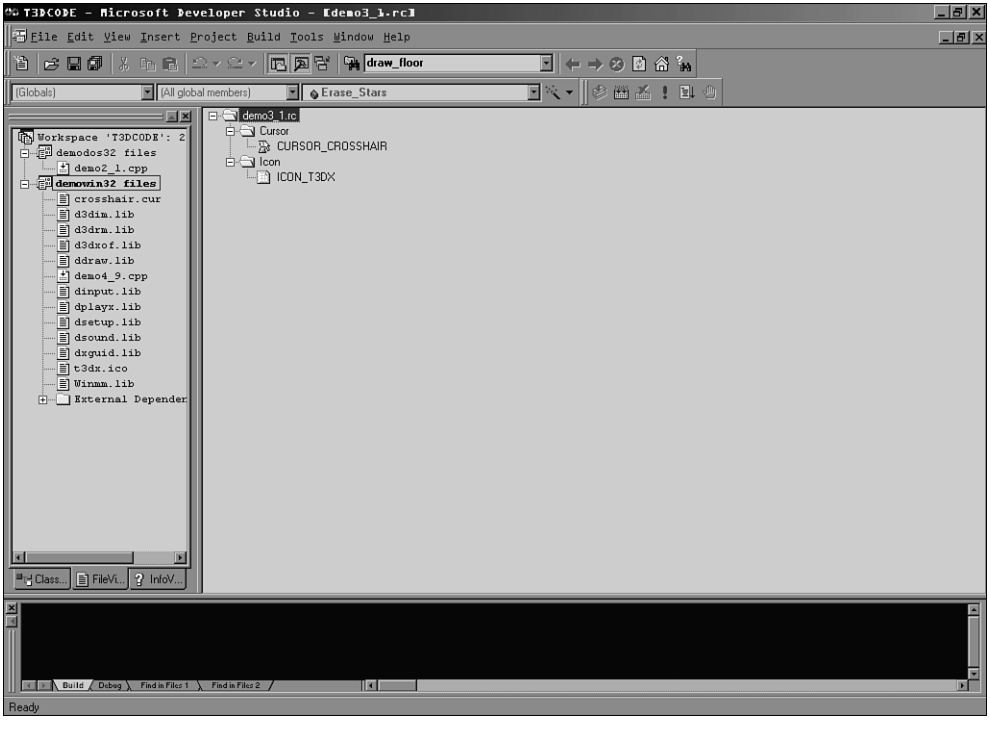


Рис. 3.6. Файл ресурса DEMO3_1.RC открыт в VC++ 6.0

Откроем в качестве эксперимента файл DEMO3_1.RC в интегрированной среде разработки (на рис. 3.6 показан результат открытия файла ресурсов в среде VC++ 6.0). Естественно, в каждом конкретном компиляторе результаты могут немного различаться, но в целом они будут схожи. Как видите, при загрузке созданного вручную .RC-файла вы не сможете его редактировать, так что все исправления, которые вы хотите внести самостоятельно, следует вносить, открыв .RC-файл как обычный текстовый. Как уже отмечалось, интегрированная среда стремится добавить ресурсы за вас — вам нужно только указать, какой именно ресурс вы хотите добавить в приложение. Но должен заметить, что читать эту книгу и прочую документацию за вас никакая интегрированная среда не сможет...

Создание ресурсов таблицы строк

Как уже упоминалось ранее, Windows поддерживает строковые ресурсы. В отличие от других ресурсов, вы можете иметь только одну таблицу строк, в которой должны находиться все строки. Кроме того, строковые ресурсы не могут использовать строковые же имена. Таким образом, все строки в вашем .RC-файле должны сопровождаться символьными константами; кроме того, должен иметься .H-файл с их определениями.

Я все еще не уверен в своем отношении к строковым ресурсам. Их использование, по сути, ничем не отличается от использования определения строк в заголовочном файле, и как при использовании строкового ресурса, так и при использовании заголовочного файла внесение каких-либо изменений требует перекомпиляции приложения. Поэтому я не очень понимаю, зачем нужны строковые ресурсы. Впрочем, строковые ресурсы могут быть размещены в DLL и основная программа при изменениях не потребует перекомпиляции (более того, можно, например, использовать .DLL-файлы для разных языков и таким образом решить проблему локализации игры. — *Прим. ред.*).

При создании таблицы строк в .RC-файле используется следующий синтаксис:

```
STRINGTABLE
{
  ID_STRING1, "string 1"
  ID_STRING2, "string 2"
.
.
}
```

Разумеется, символьные константы могут быть любыми, как и строки в кавычках. Однако есть одно ограничение: ни одна строка не должна быть длиннее 255 символов (включая константу).

Вот пример .H- и .RC-файлов, описывающих таблицу строк, которая может использоваться в игре для организации главного меню.

Содержимое .H-файла:

```
// Значения символьных констант
#define ID_STRING_START_GAME  16
#define ID_STRING_LOAD_GAME   17
#define ID_STRING_SAVE_GAME   18
#define ID_STRING_OPTIONS     19
#define ID_STRING_EXIT        20
```

Содержимое .RC-файла:

```
// Обратите внимание - таблица строк не имеет имени,
// поскольку в .RC-файле она может быть только одна
STRINGTABLE
{
ID_STRING_START_GAME "Немного поразвлечься"
ID_STRING_LOAD_GAME  "Загрузить сохраненную игру"
ID_STRING_SAVE_GAME  "Сохранить текущую игру"
ID_STRING_OPTIONS    "Установки игры"
ID_STRING_EXIT       "Выход из игры"
}
```

СОВЕТ

В строке может быть почти все, что угодно, включая спецификаторы команд printf(), такие, как %s, %d и т.п. Однако вы не можете использовать esc-последовательности типа “\n”, хотя и можете воспользоваться восьмеричными последовательностями типа \015.

После того как вы создали файл ресурсов, содержащий строковые ресурсы, для загрузки конкретной строки вы можете воспользоваться функцией LoadString(). Вот ее прототип:

```
int LoadString(HINSTANCE hInstance,// Дескриптор модуля со
                                   // строковыми ресурсами
        UINT   uID,        // Идентификатор ресурса
        LPTSTR lpBuffer,   // Адрес буфера для
                           // загрузки ресурса
        int    nBufferMax); // Размер буфера
```

Функция LoadString() возвращает количество прочитанных символов (0, если чтение ресурса было неуспешным). Вот как эта функция используется для загрузки и сохранения строк во время работы приложения:

```
// Создание места для хранения строк
char load_string[80],
   save_string[80];

// Загрузка первой строки и проверка на ошибки
if (!LoadString(hinstance, ID_STRING_LOAD_GAME,
          load_string, 80))
{
  // Произошла ошибка!
}  // if

// Загрузка второй строки и проверка на ошибки
if (!LoadString(hinstance, ID_STRING_SAVE_GAME,
          save_string, 80))
{
  // Произошла ошибка!
}  // if
```

Как обычно, hinstance представляет собой экземпляр приложения, переданный в качестве параметра в функцию WinMain().

Использование звуковых ресурсов

Большинство игр используют два типа звуков.

- Цифровые .WAV-файлы.
- Музыкальные (MIDI) .MID-файлы.

Насколько я знаю, стандартные ресурсы Windows поддерживают только .WAV-файлы, так что речь пойдет только о них. Однако, даже если Windows и не поддерживает использование MIDI-ресурсов, вы всегда можете работать с ними, как с ресурсами пользовательского типа. Я не буду рассказывать о том, как это сделать, но такая возможность есть, и, немного поработав со справочной системой, вы сможете ее реализовать.

Первое, что вам нужно, — это .WAV-файл, представляющий собой оцифрованные с некоторой частотой 8- или 16-битовые звуковые данные. Обычно частота оцифровки для звуковых эффектов в играх составляет 11, 22 или 44 kHz. Более подробно мы поговорим о звуке при изучении DirectSound.

Итак, у нас на диске есть .WAV-файл, который мы хотим добавить к приложению в качестве ресурса. Наверняка вы уже догадались, как это сделать. Для тех, кто забыл, повторю процедуру внесения ресурса в приложение еще раз.

Метод 1 — с использованием строкового имени:

```
wave_name WAVE FILENAME.WAV
```

Например:

```
BigExplosion WAVE expl1.wav
FireWeapons  WAVE fire.wav
```

Метод 2 — с использованием целочисленного идентификатора:

```
ID_WAVE WAVE FILENAME.WAV
```

Например:

```
DEATH_SOUND_ID WAVE die.wav
124      WAVE intro.wav
```

Разумеется, при использовании символьных имен вы должны создать .H-файл с определениями символьных имен; впрочем, об этом говорилось уже не раз.

Теперь попробуем разобраться, как использовать полученные ресурсы. Использование звука несколько сложнее, чем использование пиктограмм или курсоров. Чтобы воспроизвести звук после загрузки, требуется существенно больший объем работы, поэтому более подробно о звуке мы поговорим позже, а пока я только покажу, как воспроизвести звук при помощи функции PlaySound(). Вот ее прототип:

```
BOOL PlaySound(LPCSTR  pszSound, // Воспроизводимый звук
        HMODULE hmod,            // Экземпляр приложения
        DWORD  fwdSound);        // Флаги воспроизведения
```

Рассмотрим все параметры функции PlaySound().

- pszSound. Этот параметр представляет собой либо строку имени звукового ресурса, либо имя файла на диске. Кроме того, если ресурс определен при помощи символьной константы, можно использовать для вычисления данного параметра макрос MAKEINTRESOURCE().
- hmod. Экземпляр приложения, из которого осуществляется загрузка звукового ресурса. Сюда просто передается параметр hinstance функции WinMain().
- fwdSound. Этот параметр управляет загрузкой и воспроизведением звука. В табл. 3.1 приведены наиболее часто используемые значения этого параметра.

Таблица 3.1. Значения параметра fwdSound функции PlaySound()

Значение	*Описание*
SND_FILENAME	Параметр pszSound представляет собой имя файла
SND_RESOURCE	Параметр pszSound представляет собой идентификатор ресурса. Параметр hmod в этом случае должен идентифицировать экземпляр приложения, содержащего ресурс
SND_MEMORY	Звуковой файл загружен в память. Параметр pszSound при этом должен указывать на образ звука в памяти
SND_SYNC	Синхронное воспроизведение звука. Возврат из функции PlaySound() происходит только по завершении воспроизведения звука
SND_ASYNC	Асинхронное воспроизведение звука. Возврат из функции PlaySound() происходит немедленно после начала воспроизведения звука. Для прекращения асинхронного воспроизведения вызывается функция PlaySound(), параметр pszSound которой равен NULL

Окончание табл. 3.1

Значение	*Описание*
SND_LOOP	Звук воспроизводится циклически до тех пор, пока не будет вызвана функция PlaySound(), параметр pszSound которой равен NULL. Вместе с этим флагом должен быть определен флаг SND_ASYNC
SND_NODEFAULT	Звук по умолчанию не используется. Если звук не может быть найден, PlaySound() завершает работу, не воспроизводя при этом звук по умолчанию
SND_PURGE	Прекращается воспроизведение всех звуков вызывающей задачи. Если параметр pszSound не равен NULL, то прекращается воспроизведение всех экземпляров указанного звука; если параметр — NULL, прекращается воспроизведение всех звуков
SND_NOSTOP	Указанный звук будет уступать другому звуку, воспроизводимому в текущий момент. Если звук не может быть воспроизведен из-за того, что необходимые для этого ресурсы заняты для воспроизведения другого звука, функция PlaySound() немедленно возвращает значение FALSE, не воспроизводя звука
SND_NOWAIT	Если драйвер занят, функция немедленно возвращает управление без воспроизведения звука

Для того чтобы воспроизвести .WAV-ресурс с помощью функции PlaySound(), выполните ряд действий.

1. Создайте .WAV-файл и сохраните его на диске.
2. Создайте .RC-файл и связанный с ним .H-файл.
3. Скомпилируйте файл ресурса вместе с вашей программой.
4. В программе используйте вызов PlaySound() с именем звукового ресурса (или макросом MAKEINTRESOURCE() с идентификатором ресурса).

Чтобы было понятнее, рассмотрим пару примеров. Начнем с .RC-файла, в котором определены два звуковых ресурса — один с помощью строкового имени, а второй посредством символьной константы. Пусть именами соответствующих .RC- и .H-файлов будут RESOURCE.RC и RESOURCE.H. Файл RESOURCE.H содержит строку

```
#define SOUND_ID_ENERGIZE   1
```

А файл RESOURCE.RC — строки

```
#include "RESOURCE.H"

// Первый ресурс определен при помощи строкового имени
Teleporter  WAVE  teleport.wav

// Второй ресурс определен при помощи символьной константы
SOUND_ID_ENERGIZE  WAVE  energize.wav
```

В самой программе вы можете воспроизводить звуки самыми разными способами.

```
// Асинхронное воспроизведение звука
PlaySound("Teleporter",hinstance,SND_ASYNC|SND_RESOURCE);

// Асинхронное циклическое воспроизведение звука
PlaySound("Teleporter",hinstance,
      SND_ASYNC|SND_LOOP|SND_RESOURCE);
```

```
// Асинхронное воспроизведение звука
PlaySound(MAKEINTRESOURCE(SOUND_ID_ENERGIZE),hinstance,
      SND_ASYNC|SND_RESOURCE);

// Асинхронное воспроизведение звука из дискового файла
PlaySound("C:\path\filename.wav",hinstance,
      SND_ASYNC|SND_FILENAME);

// Прекращение воспроизведения звука
PlaySound(NULL,hinstance,SND_PURGE);
```

Безусловно, существует множество других опций, с которыми можно поэкспериментировать. К сожалению, пока что мы не знакомы с управляющими элементами или меню и поэтому у нас нет способов взаимодействия с демонстрационным приложением DEMO3_2.CPP, имеющимся на прилагаемом компакт-диске. Заметим, что 99% кода этого приложения занимает стандартный шаблон, а код для воспроизведения звука — всего несколько строк, аналогичных только что рассмотренным. Поэтому полный текст этого приложения в книге не приводится. Тем не менее мне бы хотелось показать вам содержимое .H- и .RC-файлов данного приложения.

Файл DEMO3_2RES.H:

```
// Определение идентификаторов звуков
#define SOUND_ID_CREATE     1
#define SOUND_ID_MUSIC      2

// Определения для пиктограмм
#define ICON_T3DX         500

// Определения для курсоров
#define CURSOR_CROSSHAIR    600
```

Файл DEMO3_2.RC:

```
#include "DEMO3_2RES.H"


// Звуковые ресурсы
SOUND_ID_CREATE   WAVE create.wav
SOUND_ID_MUSIC    WAVE techno.wav

// Пиктограмма
ICON_T3DX ICON T3DX.ICO

// Курсор
CURSOR_CROSSHAIR CURSOR CROSSHAIR.CUR
```

Обратите внимание, что в одном .RC-файле описаны ресурсы трех разных типов.

При создании демонстрационного приложения DEMO3_2.CPP я взял в качестве основы стандартный шаблон и добавил вызовы для воспроизведения звука в двух местах: при обработке сообщений WM_CREATE и WM_DESTROY. При обработке сообщения WM_CREATE запускаются два звуковых эффекта. Первый — произнесение фразы “Creating window”, а второй — циклическое воспроизведение небольшого музыкального фрагмента. При обработке сообщения WM_DESTROY воспроизведение звука останавливается.

НА ЗАМЕТКУ

Для воспроизведения первого звука я использовал флаг SND_SYNC. Таким образом я гарантировал, что первый звуковой эффект будет воспроизведен полностью, а не оборвется на средине.

Вот код обработки сообщений WM_CREATE и WM_DESTROY с вызовами функции PlaySound().

```
case WM_CREATE:
{
  // Инициализация

  // Однократное воспроизведение звука
  PlaySound(MAKEINTRESOURCE(SOUND_ID_CREATE),
        hinstance_app, SND_RESOURCE | SND_SYNC);

  // Циклическое воспроизведение звука
  PlaySound(MAKEINTRESOURCE(SOUND_ID_MUSIC),
        hinstance_app,
        SND_RESOURCE | SND_ASYNC | SND_LOOP);
   // Успешное завершение
  return(0);
} break;
.
.
.
case WM_DESTROY:
{
  // Остановка воспроизведения звука
  PlaySound(NULL, hinstance_app, SND_PURGE);

  // Закрытие приложения при помощи сообщения WM_QUIT
  PostQuitMessage(0);

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

Обратите внимание на переменную hinstance_app, которая использована в качестве дескриптора экземпляра приложения в функции PlaySound(). Это просто глобальная переменная, предназначенная для хранения значения параметра hinstance функции WinMain().

```
// Сохранение hinstance в глобальной переменной
hinstance_app = hinstance;
```

Для построения приложения вам потребуются перечисленные ниже файлы.

DEMO3_2.CPP	Главный исходный файл
DEMO3_2RES.H	Заголовочный файл с символьными именами ресурсов
DEMO3_2.RC	Сценарий ресурсов
TECHNO.WAV	Музыкальный клип
CREATE.WAV	Звуковое сообщение о создании окна
WINMM.LIB	Расширения библиотеки мультимедиа Windows. Этот файл находится в каталоге LIB\ вашего компилятора и должен быть добавлен к проекту
MMSYSTEM.H	Заголовочный файл для WINMM.LIB. Этот файл включен как в DEMO3_2.CPP, так и во все остальные демонстрационные программы. Все, что требуется от вас, — чтобы стандартный заголовочный файл Win32 находился в одном из каталогов, где компилятор осуществляет поиск заголовочных файлов

Использование компилятора для создания .RC-файлов

Большинство компиляторов, создающих приложения Windows, включают богатую интегрированную среду разработки, такую, например, как Microsoft Visual Development Studio или подобную. Каждая из этих интегрированных сред содержит один или несколько инструментов для создания различных ресурсов, сценариев ресурсов и связанных с ними заголовочных файлов автоматически и/или с использованием технологии "перетащить и отпустить".

Единственная проблема при использовании этого инструментария состоит в том, что вы должны его изучить. Кроме того, создаваемые интегрированной средой .RC-файлы представляют собой вполне удобочитаемые текстовые файлы, но в них содержится масса директив препроцессора и макросов, которые компилятор добавляет для автоматизации и упрощения выбора констант и обеспечения интерфейса с MFC.

Поскольку в настоящее время я пользуюсь Microsoft VC++ 6.0, я вкратце расскажу о ключевых элементах поддержки работы с ресурсами в VC++ 6.0. В нем имеется два пути добавления ресурсов в ваш проект.

Первый метод состоит в использовании команды меню File⇨New. На рис. 3.7 показано появляющееся при этом диалоговое окно. При добавлении ресурса типа пиктограммы, курсора или растрового изображения интегрированная среда автоматически запускает Image Editor (показанный на рис. 3.4). Это очень простой графический редактор, который вы можете использовать для создания собственных курсоров и пиктограмм. При добавлении ресурса меню (о котором мы поговорим в следующем разделе) будет вызван редактор меню.

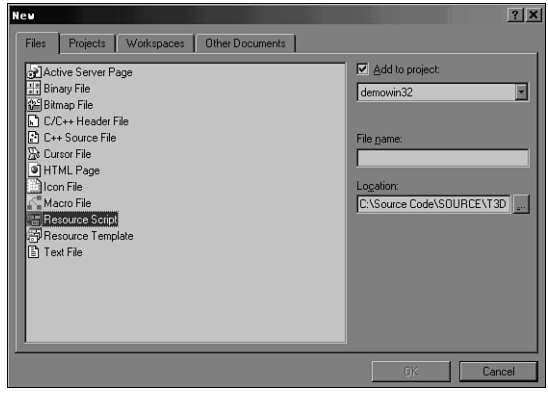


Рис. 3.7. Добавление ресурса в проект VC++ 6.0 при помощи команды меню File⇨New

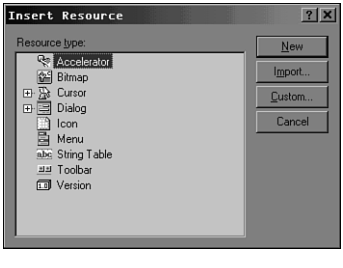


Рис. 3.8. Добавление ресурса в проект VC++ 6.0 при помощи команды меню Insert⇨Resource

Второй метод более гибок и позволяет работать с ресурсами любого типа, в то время как первый поддерживает только строго определенное множество типов ресурсов. Для добавления в ваш проект ресурса любого типа можете воспользоваться командой главного меню Insert⇨Resource. Появляющееся при этом диалоговое окно показано на рис. 3.8. Однако этот метод выполняет некоторые скрытые действия. При добавлении ресурса вы должны внести его в сценарий ресурса — ведь так? И если ваш проект еще не имеет этого файла, интегрированная среда сгенерирует его для вас и назовет SCRIPT*.RC. Кроме того, оба метода генерируют (и/или изменяют) файл RESOURCE.H, в котором содержатся символьные имена ресурсов, значения идентификаторов и т.п.

Я бы хотел рассказать об использовании интегрированной среды для работы с ресурсами подробнее, но материала здесь вполне хватит на целую главу, если не вообще на книгу. Так что все, что я могу сделать для вас, — это посоветовать почитать документацию к вашему компилятору. В этой книге мы будем использовать не так уж много ресурсов, так что той информации, которую вы получили, вам должно хватить. Перейдем лучше к более сложному типу ресурсов — меню.

Работа с меню

Меню являются одним из наиболее нужных элементов в программировании в Windows и одним из наиболее важных способов взаимодействия пользователя с программой. Таким образом, знание того, как создаются и работают меню, совершенно необходимо при программировании в Windows. Это знание требуется и при написании игр — например, при создании вспомогательного инструментария или при запуске игры с помощью начального окна. Словом, примите на веру, что знания о том, как создать и загрузить меню и как работать с ним, никогда не будут лишними.

Создание меню

Вы можете создать меню и все связанные с ним файлы с помощью редактора меню интегрированной среды разработки, но того же результата можно добиться и вручную (именно так мы и будем поступать, поскольку я не знаю, какой именно компилятор вы используете). Впрочем, при написании реальных приложений вы, возможно, будете пользоваться редактором меню, поскольку меню бывают слишком сложными для создания вручную.

Итак, приступим к созданию меню. Они во многом похожи на другие ресурсы, с которыми вы уже работали. Они размещаются в .RC-файле, а связанные с ними символьные константы — в .H-файле (для меню все они должны представлять собой целочисленные идентификаторы, за исключением лишь имени меню). Вот синтаксис описания меню в .RC-файле:

```
MENU_NAME MENU DISCARDABLE
{  // При желании вы можете использовать
   // вместо "{" ключевое слово BEGIN

   // Определение меню

}  // При желании вы можете использовать
   // вместо "}" ключевое слово END
```

MENU_NAME может быть строковым именем или символьной константой, а ключевое слово DISCARDABLE — необходимый рудимент. Выглядит довольно просто, но пока пустует само определение меню.

Перед тем как приводить код определения меню и подменю, следует определиться с терминологией. Рассмотрим рис. 3.9. На нем показаны два меню верхнего уровня — File

и Help. Меню File содержит команды Open, Close, Save и Exit, а меню Help — только одну команду About. Таким образом, здесь представлена ситуация с меню верхнего уровня и пунктами в них. Однако внутри меню могут быть вложены другие меню, образуя так называемые *каскадные меню*. Я не буду их использовать, но теория проста: вместо описания пункта меню используется определение меню. Это действие может быть рекурсивным до бесконечности.

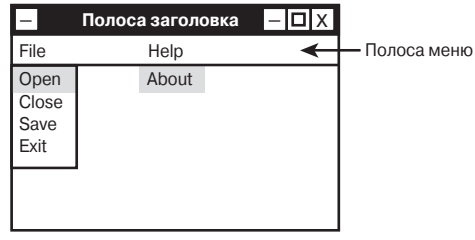


Рис. 3.9. Полоса меню с двумя подменю

Теперь посмотрим, как же реализуется показанное на рис. 3.9 меню.

```
MainMenu MENU DISCARDABLE
{
  POPUP "File"
  {
    MENUITEM "Open",  MENU_FILE_ID_OPEN
    MENUITEM "Close", MENU_FILE_ID_CLOSE
    MENUITEM "Save",  MENU_FILE_ID_SAVE
    MENUITEM "Exit",  MENU_FILE_ID_EXIT
  } // POPUP

  POPUP "Help"
  {
    MENUITEM "About", MENU_HELP_ABOUT
  } // POPUP

} // Конец описания меню
```

Рассмотрим определение меню раздел за разделом. Начнем с того, что имя приведенного меню — MainMenu. Это может быть как целочисленный идентификатор, так и строковое имя (исходя из того, что все идентификаторы у меня состоят из прописных букв, можно сделать вывод, что в данном случае это строчное имя). Далее описаны два меню верхнего уровня. Каждое описание начинается с ключевого слова POPUP, указывающего, что далее определяется меню с указанным именем и его пунктами.

Имя меню должно быть заключено в кавычки, а определение меню — содержаться между фигурными скобками {} (вместо них можно использовать ключевые слова BEGIN и END соответственно — программисты на Pascal должны оценить эту возможность).

В блоке определения меню располагаются все пункты данного меню. Для определения пункта используется ключевое слово MENUITEM со следующим синтаксисом:

```
MENUITEM "name", MENU_ID
```

И, конечно, в соответствующем .H-файле должны быть определены используемые при описании меню символьные константы наподобие приведенных.

```
// Определения для меню File
#define MENU_FILE_ID_OPEN   1000
#define MENU_FILE_ID_CLOSE  1001
#define MENU_FILE_ID_SAVE   1002
#define MENU_FILE_ID_EXIT   1003

// Определения для меню Help
#define MENU_HELP_ABOUT     2000
```

СОВЕТ

Обратите внимание на значения идентификаторов. Для начального значения первого меню верхнего уровня я выбрал 1000 и увеличивал это значение на 1 для каждого последующего пункта. При переходе к следующему меню верхнего уровня было выбрано увеличенное на 1000 значение предыдущего меню (т.е. все меню верхнего уровня отличаются от предыдущего на 1000, а пункты внутри меню — на 1). Это соглашение легко реализуется и неплохо работает, так что непременно воспользуйтесь им.

Я не определял значение MainMenu, так как мы уже решили, что это строковое имя, а не символьный идентификатор. Но это, конечно, не обязательно и в .H-файле вполне можно поместить строку

```
#define MainMenu  100
```

После этого для обращения к меню мне просто пришлось бы использовать макрос MAKEINTRESOURCE(MainMenu) или MAKEINTRESOURCE(100).

СЕКРЕТ

Наверняка в приложениях Windows вы видели пункты меню с горячими клавишами, которые можно использовать вместо выбора меню верхнего уровня или пунктов меню с помощью мыши. Это достигается путем использования символа &, который должен быть размещен перед выбранным вами в качестве горячей клавиши символом в описании POPUP или MENUITEM. Так, например, описание

```
MENUITEM "E&xit", MENU_FILE_ID_EXIT
```

делает горячей клавишу x, а описание

```
POPUP "&File"
```

делает <Alt+F> горячими клавишами для данного меню.

Теперь, когда вы знаете, как создать меню, самое время узнать о том, как им воспользоваться.

Загрузка меню

Существует несколько способов присоединить меню к окну. Вы можете связать одно меню со всеми окнами посредством указания его в классе Windows либо присоединять к каждому создаваемому меню свое окно. Для начала посмотрим, как связать одно меню со всеми создаваемыми окнами путем определения его в классе Windows.

Для этого при определении класса Windows мы просто присваиваем полю lpszMenuName имя ресурса меню

```
winclass.lpszMenuName = "MainMenu";
```

или, если MainMenu представляет собой символьный идентификатор, можем воспользоваться соответствующим макросом:

```
winclass.lpszMenuName = MAKEINTRESOURCE(MainMenu);
```

Как видите, никаких проблем. Ну или почти никаких, если не считать того, что все создаваемые окна будут иметь одно и то же меню. Однако этой неприятности можно избежать,

если назначать меню окну при его создании, передавая ему дескриптор меню, который можно получить при помощи функции LoadMenu(), имеющей следующий прототип:

```
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);
```

При успешном завершении функция возвращает дескриптор меню, который вы можете использовать, например, в функции CreateWindowEx():

```
// Создание окна
if (!(hwnd = CreateWindowEx(NULL,
                  WINDOW_CLASS_NAME,
                  "Sound Resource Demo",
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  0, 0, 400, 400,
                  NULL,
                  LoadMenu(hinstance, "MainMenu"),
                  hinstance,
                  NULL)))
return(0);
```

Если MainMenu представляет собой символьную константу, то выделенная полужирным шрифтом строка должна выглядеть как

```
LoadMenu(hinstance, MAKEINTRESOURCE(MainMenu)),
```

НА ЗАМЕТКУ

Я наверняка уже замучил вас упоминаниями о том, как надо работать со строковыми именами, а как — с символьными константами. Но если учесть, что это наиболее распространенная ошибка у программистов в Windows, то мое занудство не покажется вам таким странным.

Разумеется, в .RC-файле может быть определено множество различных меню и каждому окну вы можете назначить собственное.

Последний метод назначения меню — использование функции SetMenu().

```
BOOL SetMenu(HWND  hWnd,  // Дескриптор окна
      HMENU hMenu); // Дескриптор меню
```

В качестве параметров данная функция принимает дескриптор окна и дескриптор меню, полученный вызовом функции LoadMenu(). Новое меню заменяет старое присоединенное к окну меню, если таковое имелось. Итак, вот как может выглядеть код с использованием функции SetMenu():

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

```
// Регистрация класса
if (!RegisterClassEx(&winclass))
   return(0);

// Создание окна
if (!(hwnd = CreateWindowEx(NULL,
                  WINDOW_CLASS_NAME,
                  "Menu Resource Demo",
                  WS_OVERLAPPEDWINDOW|WS_VISIBLE,
                   0,0,
                  400,400,
                  NULL,
                 NULL,   // Имя меню
                  hinstance,
                  NULL)))
return(0);

// Загрузка ресурса меню
HMENU hmenuhandle = LoadMenu(hinstance, "MainMenu");

// Назначение меню окну
SetMenu(hwnd, hmenuhandle);
```

Этот код взят из имеющейся на прилагаемом компакт-диске демонстрационной программы DEMO3_3.CPP, работа которой показана на рис. 3.10.

Рис. 3.10. Работа программы DEMO3_3.EXE

В этом проекте нас в первую очередь интересуют файл ресурса и заголовочный файл, представленные ниже.

Содержимое файла DEMO3_3RES.H:

```
// Определения для меню File
#define MENU_FILE_ID_OPEN   1000
#define MENU_FILE_ID_CLOSE  1001
#define MENU_FILE_ID_SAVE   1002
#define MENU_FILE_ID_EXIT   1003

// Определения для меню Help
#define MENU_HELP_ABOUT     2000
```

Содержимое файла DEMO3_3.RC:

```
#include "DEMO3_3RES.H"

MainMenu MENU DISCARDABLE
{
POPUP "File"
  {
    MENUITEM "Open", MENU_FILE_ID_OPEN
    MENUITEM "Close", MENU_FILE_ID_CLOSE
    MENUITEM "Save", MENU_FILE_ID_SAVE
    MENUITEM "Exit", MENU_FILE_ID_EXIT
  } // popup

POPUP "Help"
  {
    MENUITEM "About", MENU_HELP_ABOUT
  } // popup

} // menu
```

Для компиляции проекта требуются следующие файлы:

DEMO3_3.CPP	Главный исходный файл
DEMO3_3RES.H	Заголовочный файл с символьными константами
DEMO3_3.RC	Файл сценария ресурса

Попробуйте поработать с файлом DEMO3_3.CPP — измените меню, добавьте новые меню верхнего уровня, попытайтесь создать каскадные меню (подсказка: просто замените MENUITEM на POPUP с одним или несколькими описаниями MENUITEM).

Обработка сообщений меню

Главная проблема приложения DEMO3_3.EXE состоит в том, что при выборе того или иного пункта меню ничего не происходит! Дело в том, что вы пока не знаете, как определить, что выбран тот или иной пункт меню, и как на него отреагировать. Этому вопросу и посвящен данный подраздел.

При выборе какого-либо пункта меню генерирует сообщение, как показано на рис. 3.11.

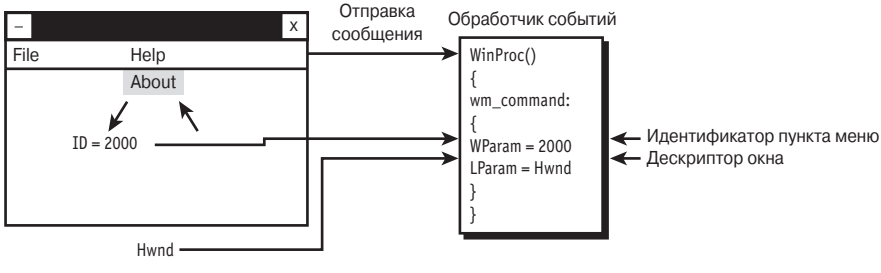


Рис. 3.11. Сообщение при выборе пункта меню

Интересующее нас сообщение посылается при выборе пункта меню и отпускании кнопки мыши (именно это и называется *выбором* пункта меню). При этом обработчику событий окна, с которым связано это меню, посылается сообщение WM_COMMAND; при этом конкретный выбранный пункт меню и другие данные передаются как приведенные ниже параметры сообщения.

msg	WM_COMMAND
lparam	Дескриптор окна — отправителя сообщения
wparam	Идентификатор выбранного пункта меню

СОВЕТ

Технически вы должны выделить нижнее значение типа WORD из параметра wparam при помощи макроса LOWORD(). Этот макрос является частью стандартных включаемых файлов, так что вы автоматически получаете к нему доступ.

Итак, все, что вы должны сделать, — это использовать конструкцию switch() с параметром wparam, которая определяет, какие именно действия выполнять при выборе того или иного пункта меню. Вот как это можно было бы сделать в демонстрационном примере DEMO3_3.CPP:

```
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam)
{
// Основной обработчик сообщений
PAINTSTRUCT   ps;   // Используется в WM_PAINT
HDC           hdc;  // Дескриптор контекста устройства
// Какое сообщение получено
switch(msg)
  {
  case WM_CREATE:
    {
    // Выполнение инициализации

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

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

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

  case WM_DESTROY:
    {
    // Завершение работы приложения
    PostQuitMessage(0);

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

  case WM_COMMAND:
    {
```

```
    switch(LOWORD(wparam))
      {
      case MENU_FILE_ID_OPEN:
        {
        // Выполнение соответствующих действий
        } break;
      case MENU_FILE_ID_CLOSE:
        {
        // Выполнение соответствующих действий
        } break;
      case MENU_FILE_ID_SAVE:
        {
        // Выполнение соответствующих действий
        } break;
      case MENU_FILE_ID_EXIT:
        {
        // Выполнение соответствующих действий
        } break;
      case MENU_HELP_ABOUT:
        {
        // Выполнение соответствующих действий
        } break;
      default: break;
      }
    } break; // WM_COMMAND

  default:break;

  } // switch

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

} // WinProc
```

Это так просто, что все время ждешь подвоха. Конечно, имеются и другие сообщения, которые управляют самими меню и их пунктами, но о них вы можете узнать в справочной системе Win32 SDK. Со своей стороны могу сказать, что мне очень редко требуется нечто большее, чем просто знать, был ли выбран пункт меню и какой именно.

В качестве конкретного примера работы с меню я разработал программу DEMO3_4.CPP, которая с использованием меню позволяет выйти из программы, воспроизвести один из четырех различных звуковых эффектов и вывести диалоговое окно About. Файл ресурсов этого приложения содержит звуковые ресурсы, пиктограмму и курсор.

Файл DEMO3_4RES.H:

```
// Определения для звуковых ресурсов
#define SOUND_ID_ENERGIZE  1
#define SOUND_ID_BEAM      2
#define SOUND_ID_TELEPORT  3
#define SOUND_ID_WARP  4

// Определения пиктограммы и курсора
#define ICON_T3DX      100
#define CURSOR_CROSSHAIR 200
```

```
// Определения для меню верхнего уровня File
#define MENU_FILE_ID_EXIT            1000

// Определения для воспроизведения звуков
#define MENU_PLAY_ID_ENERGIZE        2000
#define MENU_PLAY_ID_BEAM            2001
#define MENU_PLAY_ID_TELEPORT        2002
#define MENU_PLAY_ID_WARP            2003

// Определения для меню верхнего уровня Help
#define MENU_HELP_ABOUT              3000
```

Файл DEMO3_4.RC:

```
#include "DEMO3_4RES.H"

// Пиктограмма и курсор
ICON_T3DX      ICON   t3dx.ico
CURSOR_CROSSHAIR CURSOR crosshair.cur

// Звуковые ресурсы
SOUND_ID_ENERGIZE   WAVE energize.wav
SOUND_ID_BEAM       WAVE beam.wav
SOUND_ID_TELEPORT   WAVE teleport.wav
SOUND_ID_WARP       WAVE warp.wav

// Меню
SoundMenu MENU DISCARDABLE
{
POPUP "&File"
  {
  MENUITEM "E&xit",  MENU_FILE_ID_EXIT
  } // POPUP

POPUP "&PlaySound"
 {
  MENUITEM  "Energize!",          MENU_PLAY_ID_ENERGIZE
  MENUITEM  "Beam Me Up",         MENU_PLAY_ID_BEAM
  MENUITEM  "Engage Teleporter",    MENU_PLAY_ID_TELEPORT
  MENUITEM  "Quantum Warp Teleport", MENU_PLAY_ID_WARP
 } // POPUP

POPUP "Help"
  {
  MENUITEM "About",  MENU_HELP_ABOUT
  } // POPUP

} // MENU
```

Рассмотрим теперь код загрузки каждого из ресурсов. Сначала загружаются меню, пиктограмма и курсор:

```
winclass.hIcon       = LoadIcon(hinstance,
                MAKEINTRESOURCE(ICON_T3DX));
```

```
winclass.hCursor      = LoadCursor(hinstance,
                MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.lpszMenuName  = "SoundMenu";
winclass.hIconSm      = LoadIcon(hinstance,
                MAKEINTRESOURCE(ICON_T3DX));
```

Теперь самое интересное, а именно обработка сообщения WM_COMMAND, где и воспроизводятся звуки. Я привожу здесь только обработчик сообщения WM_COMMAND, так как со всей WinProc вы уже достаточно хорошо знакомы.

```
case WM_COMMAND:
   {
    switch(LOWORD(wparam))
        {
        // Обработка меню File
        case MENU_FILE_ID_EXIT:
           {
           // Закрытие окна
           PostQuitMessage(0);
           } break;

        // Обработка меню Help
        case MENU_HELP_ABOUT:
           {
           // Вывод диалогового окна
           MessageBox(hwnd, "Menu Sound Demo",
                  "About Sound Menu",
                   MB_OK | MB_ICONEXCLAMATION);
           } break;

        // Обработка выбора звука
        case MENU_PLAY_ID_ENERGIZE:
           {
           // Воспроизведение звука
           PlaySound(
              MAKEINTRESOURCE(SOUND_ID_ENERGIZE),
              hinstance_app, SND_RESOURCE|SND_ASYNC
                 );
           } break;

        case MENU_PLAY_ID_BEAM:
           {
           // Воспроизведение звука
           PlaySound(MAKEINTRESOURCE(SOUND_ID_BEAM),
                  hinstance_app,
                  SND_RESOURCE|SND_ASYNC);
           } break;

        case MENU_PLAY_ID_TELEPORT:
           {
           // Воспроизведение звука
           PlaySound(
              MAKEINTRESOURCE(SOUND_ID_TELEPORT),
              hinstance_app, SND_RESOURCE|SND_ASYNC);
```

```
        } break;

      case MENU_PLAY_ID_WARP:
        {
        // Воспроизведение звука
        PlaySound(MAKEINTRESOURCE(SOUND_ID_WARP),
              hinstance_app,
              SND_RESOURCE|SND_ASYNC);
        } break;

      default: break;

    }

  } break; // WM_COMMAND
```

Вот и все, что я хотел рассказать вам о ресурсах вообще и меню в частности.

Как видите, ресурсы довольно легко создавать и использовать. А мы пойдем дальше и рассмотрим более сложное сообщение WM_PAINT и основы GDI.

Введение в GDI

Пока что с маленьким кусочком GDI мы сталкивались только при обработке сообщения WM_PAINT. GDI (что расшифровывается как Graphics Device Interface — интерфейс графического устройства) представляет собой технологию вывода изображения в Windows без использования DirectX. Пока что мы не сталкивались с этой ключевой для нас задачей — ведь самая главная часть любой видеоигры та, где осуществляется вывод на экран. Можно сказать, что видеоигра представляет собой логику, которая управляет видеовыводом. В этом разделе мы вернемся к сообщению WM_PAINT, рассмотрим некоторые базовые видеоконцепции и научимся выводить в окне текст. Более детальному изучению GDI посвящена следующая глава.

Понимание работы сообщения WM_PAINT очень важно для стандартной графики GDI и программирования в Windows, поскольку большинство программ Windows осуществляют вывод именно таким способом — посредством этого сообщения. При использовании DirectX это не так, поскольку в этом случае вывод осуществляется самим DirectX (или, точнее, его частями — DirectDraw и Direct3D), но в любом случае писать приложения Windows без знания GDI невозможно.

Еще раз о сообщении WM_PAINT

Сообщение WM_PAINT посылается процедуре вашего окна, когда его клиентская область требует перерисовки. До сих пор мы практически не обрабатывали это сообщение, используя следующий код:

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

Взгляните на рис. 3.12, на котором поясняется работа сообщения WM_PAINT. Когда окно перемещается, изменяет размер или перекрывается другими окнами, вся клиент-

ская область окна (или некоторая ее часть) требует перерисовки. Когда это происходит, окну посылается сообщение WM_PAINT.

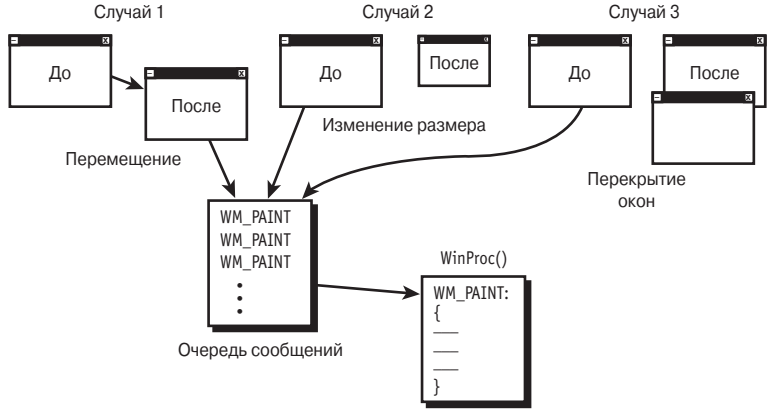


Рис. 3.12. Сообщение WM_PAINT

В случае использования представленного ранее кода вызовы BeginPaint() и EndPaint() выполняют две задачи: делают действительной (актуальной) клиентскую область окна и заполняют фон вашего окна с использованием определенной в классе окна фоновой кисти.

Теперь, если вы хотите, то можете добавить вывод графики между вызовами BeginPaint() и EndPaint(). Здесь есть только одна проблема: вы получите доступ только к той части клиентской области окна, которая требует перерисовки. Координаты ставшего недействительным (неактуальным) прямоугольника хранятся в поле rcPaint структуры ps типа PAINSTRUCT, которую возвращает вызов BeginPaint().

```
typedef struct tagPAINTSTRUCT
{
  HDC  hdc;
  // Контекст графического устройства
  BOOL fErase;
  // Если поле равно TRUE, вы должны перерисовать фон
  RECT rcPaint;
  // Прямоугольник с недействительной областью
  BOOL fRestore;          // Для внутреннего использования
  BOOL fIncUpdate;        // Для внутреннего использования
  BYTE rgbReserved[32]; // Для внутреннего использования
} PAINTSTRUCT;
```

Чтобы освежить вашу память, напомню определение структуры RECT:

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

Другими словами, возвращаясь к рис. 3.12: например, все окно имеет размер 400×400, но перерисовать мы должны только меньшую область — с координатами от 300×300 до

400×400. Таким образом, контекст графического устройства, возвращаемый вызовом BeginPaint(), корректен только для этой области окна размером 100×100. Безусловно, это создает неудобства, если вы хотите иметь доступ ко всей клиентской области окна.

Решение этой проблемы — получение доступа к контексту графического устройства окна непосредственно, а не как к части сообщения о перерисовке клиентской области. Вы всегда можете получить графический контекст окна с помощью функции GetDC():

```
HDC GetDC(HWND hwnd); // Дескриптор окна
```

Вы просто передаете функции дескриптор окна, к контексту графического устройства которого хотите получить доступ, и функция возвращает вам его дескриптор. Если функция завершается неудачно, она вернет значение NULL. По окончании работы с контекстом графического устройства вы должны освободить его с помощью вызова функции ReleaseDC().

```
int ReleaseDC( HWND hwnd, // Дескриптор окна
        HDC  hDC);        // Дескриптор контекста устройства
```

Эта функция получает в качестве параметров дескриптор окна и дескриптор контекста устройства, полученный ранее вызовом GetDC().

НА ЗАМЕТКУ

Вообще говоря, дескриптор контекста устройства может означать несколько устройств вывода информации, например, кроме экрана, это может быть и принтер. Хотя я говорю о контексте графического устройства, аббревиатура HDC означает просто дескриптор контекста устройства, и этот тип может использоваться не только для графических устройств. Однако в нашей книге эти понятия взаимозаменяемы, так что беспокоиться об этой тонкости вам не следует.

Вот как работает пара функций GetDC() — ReleaseDC():

```
HDC gdc = NULL; // Эта переменная будет хранить контекст
                // графического устройства

// Получение контекста графического устройства окна
if (!(gdc = GetDC(hwnd)))
  error();

// Использование gdc для работы с графикой
// (пока что вы не знаете, как это осуществить)

// Освобождение контекста устройства
ReleaseDC(hwnd, gdc);
```

Конечно, пока что вы не знаете, как именно выводить графику в окне, но со временем я научу вас этому. Главное, что теперь у вас есть еще один способ обработки сообщения WM_PAINT. Однако при этом возникает и небольшая проблема. Дело в том, что при работе GetDC() — ReleaseDC() Windows не знает, что вы восстановили и тем самым сделали действительной клиентскую область окна, поэтому, используя пару GetDC() — ReleaseDC() вместо пары BeginPaint() — EndPaint(), мы получили новую проблему!

Пара BeginPaint() — EndPaint() отправляет Windows сообщение, указывающее, что необходимая перерисовка окна выполнена (даже если вы не сделали ни одного графического вызова). Соответственно, больше сообщений WM_PAINT от Windows не поступает (окно полностью перерисовано). Но если использовать пару функций GetDC() — ReleaseDC() в обработчике WM_PAINT, то это сообщение будет посылаться окну постоянно. Почему? Потому что вы должны объявить, что окно стало действительным.

Конечно, после работы функций GetDC() — ReleaseDC() можно вызвать пару BeginPaint() — EndPaint(), но неэффективность данного пути очевидна. К счастью, Windows предлагает для этого специальную функцию ValidateRect():

```
BOOL ValidateRect(
  HWND hWnd,              // Дескриптор окна
  CONST RECT * lpRect); // Адрес координат прямоугольника,
                          // объявляемого действительным
```

Для объявления окна действительным следует передать функции его дескриптор наряду с указанием области, которая становится действительной. Таким образом, использование пары GetDC() — ReleaseDC() в обработчике WM_PAINT должно иметь следующий вид:

```
PAINTSTRUCT ps;
HDC      hdc;
RECT     rect;

case WM_PAINT:
{
  hdc = GetDC(hwnd);
  // Здесь выполняется перерисовка окна
  ReleaseDC(hwnd,hdc);

  // Получение прямоугольника окна
  GetClientRect(hwnd,&rect);
  // Объявление окна действительным
  ValidateRect(hwnd,&rect);

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

НА ЗАМЕТКУ

Обратите внимание на вызов GetClientRect(). Все, что делает данная функция, — возвращает координаты клиентского прямоугольника. Запомните, что, поскольку окно может перемещаться по экрану, имеется два множества координат: *координаты окна* и *клиентские координаты*. Координаты окна даются относительно экрана, а клиентские координаты — относительно верхнего левого угла окна. На рис. 3.13 все это изображено более понятно.

Вы можете спросить: "Неужели все должно быть так сложно?" Ну конечно — это же Windows! :-) Вспомните — ведь все это происходит из-за того, что нам понадобилось иметь возможность перерисовать все окно, а не только ставшую недействительной его часть.

Впрочем, сейчас я продемонстрирую вам один хитрый трюк. Если в обработчике сообщения WM_PAINT указать, что все окно стало недействительным, то поле rcPaint структуры PAINTSTRUCT, возвращаемой функцией BeginPaint(), и связанный с ним контекст устройства дадут вам доступ ко всей клиентской области окна. Для того же, чтобы указать, что все окно стало недействительным, можно воспользоваться функцией InvalidateRect().

```
BOOL InvalidateRect(
  HWND      hwnd,       // Дескриптор окна
  CONST RECT * lpRect,  // Адрес координат прямоугольника
  BOOL      bErase);    // Флаг удаления фона
```

Если параметр bErase равен TRUE, вызов BeginPaint() заполнит фон окна соответствующей кистью; в противном случае фон не перерисовывается.

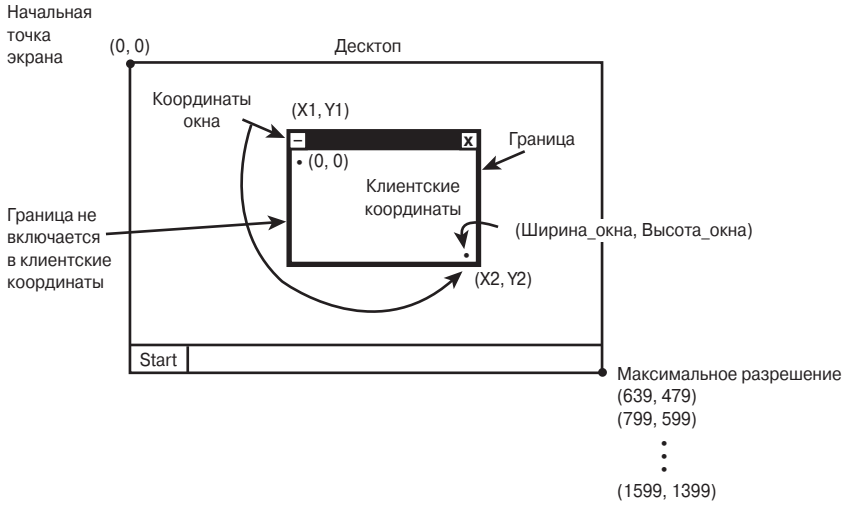


Рис. 3.13. Координаты окна и клиентские координаты

Вызвать эту функцию можно непосредственно перед вызовом BeginPaint(), которая после этого предоставит доступ к области, представляющей собой объединение недействительной области, переданной Windows при сообщении WM_PAINT, и области, определенной вами при вызове InvalidateRect(). В большинстве случаев в качестве второго параметра этой функции используется значение NULL, которое объявляет недействительным все окно полностью. Итак, вот каким получается код обработчика сообщения WM_PAINT:

```
PAINTSTRUCT ps;
HDC      hdc;

case WM_PAINT:
{
  // Объявляем недействительным все окно
  InvalidateRect(hwnd,NULL,FALSE);

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

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

В этой книге в обработчике WM_PAINT я буду использовать пару BeginPaint() — EndPaint(), а в прочих местах — пару GetDC() — ReleaseDC(). А теперь рассмотрим работу графических функций, чтобы вы могли по крайней мере вывести в окно текст.

Основы видеовывода и цвета

Пришла пора рассмотреть некоторые концепции и терминологию, связанные с графикой и цветом на персональных компьютерах. Начнем с некоторых определений.

- Пиксель — отдельный адресуемый элемент растрового изображения.
- Разрешение — количество пикселей, поддерживаемых видеокартой, например 640×480, 800×600 и т.д. Чем выше разрешение, тем качественнее изображение, но

тем большее количество памяти для него требуется. В табл. 3.2 перечислены некоторые из наиболее распространенных разрешений и их требования к памяти.

- Глубина цвета — количество битов или байтов, представляющих каждый пиксель на экране (bpp — bits per pixel, бит на пиксель). Например, если каждый пиксель представлен 8 битами (один байт), то на экране могут поддерживаться только 256 цветов, так как $2^8 = 256$. Если каждый пиксель состоит из двух байт, количество поддерживаемых цветов становится равным $2^{16} = 16384$. Чем больше глубина цвета изображения, тем более оно детально, однако тем большее количество памяти для него требуется. Кроме того, 8-битовые цвета обычно представлены палитрой (что это означает, вы вскоре узнаете); 16-битовые режимы обычно называют высокоцветными (high color), а 24- и 32-битовые — истинно- и ультрацветными соответственно (true color и ultra color).
- Чересстрочный вывод — компьютер выводит изображения с помощью сканирующей электронной пушки построчно (так называемая растеризация). Телевизионный стандарт выводит два кадра для одного изображения. Один кадр состоит из всех нечетных строк, второй — из четных. Когда эти два кадра быстро сменяют друг друга, наш глаз воспринимает их как единое изображение. Этот способ применим для вывода движущихся изображений и не подходит для статических. Однако некоторые карты способны поддерживать режимы с высоким разрешением только при чересстрочном выводе. При его использовании обычно заметно мерцание или подрагивание экрана.
- Видеопамять (VRAM) — количество памяти, имеющейся на видеокарте для представления изображений на экране.
- Частота обновления — количество обновлений изображения на экране за одну секунду, измеряемое в Hz или fps. В настоящее время 60 Hz рассматривается как минимально приемлемая величина, а ряд мониторов и видеокарт могут работать с частотами свыше 100 Hz.
- 2D-ускорение — аппаратная возможность видеокарты, помогающая Windows и/или DirectX в работе с двухмерными операциями, такими, как вывод растровой графики, линий, окружностей, текста и т.п.
- 3D-ускорение — аппаратная возможность видеокарты, помогающая Windows и/или DirectX/Direct3D при работе с трехмерной графикой.

Все рассмотренные элементы показаны на рис. 3.14.

Таблица 3.2. Разрешения и требования к памяти

Разрешение	*Бит на пиксель*	*Память (min — max)*
320×200*	8	64K
320×240*	8	64K
640×480	8, 16, 24, 32	307K — 1.22M
800×600	8, 16, 24, 32	480K — 1.92M
1024×768	8, 16, 24, 32	768K — 3.14M
1280×1024	8, 16, 24, 32	1.31M — 5.24M
1600×1200	8, 16, 24, 32	1.92M — 7.68M

*Эти режимы рассматриваются как режимы Mode X и могут не поддерживаться вашей видеокартой.

Конечно, в табл. 3.2 приведены только некоторые из возможных видеорежимов и глубин цветов. Ваша видеокарта может поддерживать гораздо большее количество режимов (или не поддерживать некоторые из перечисленных). Главное — понимать, что 2 или 4 Мбайт видеопамяти — это вовсе не много. К счастью, большинство игр с использованием DirectX, которые вы будете разрабатывать, будут работать в режимах 320×240 или 640×480, которые видеокарта с памятью 2 Мбайт вполне способна поддержать.

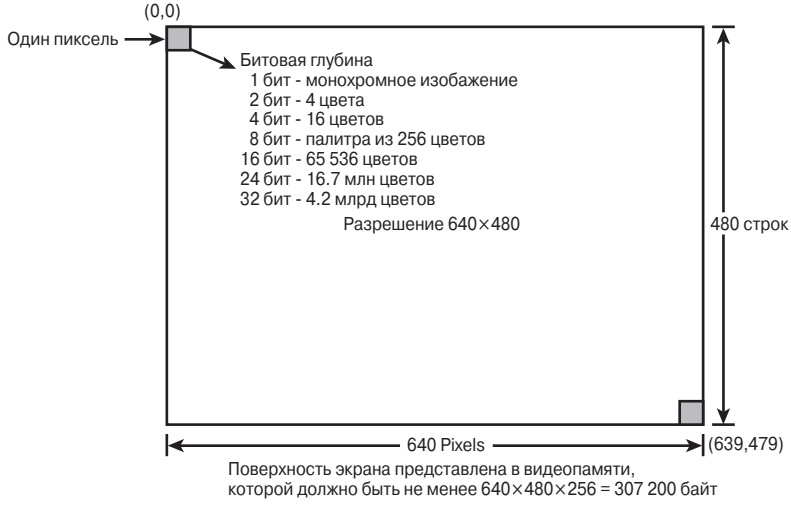


Рис. 3.14. Механизм видеовывода

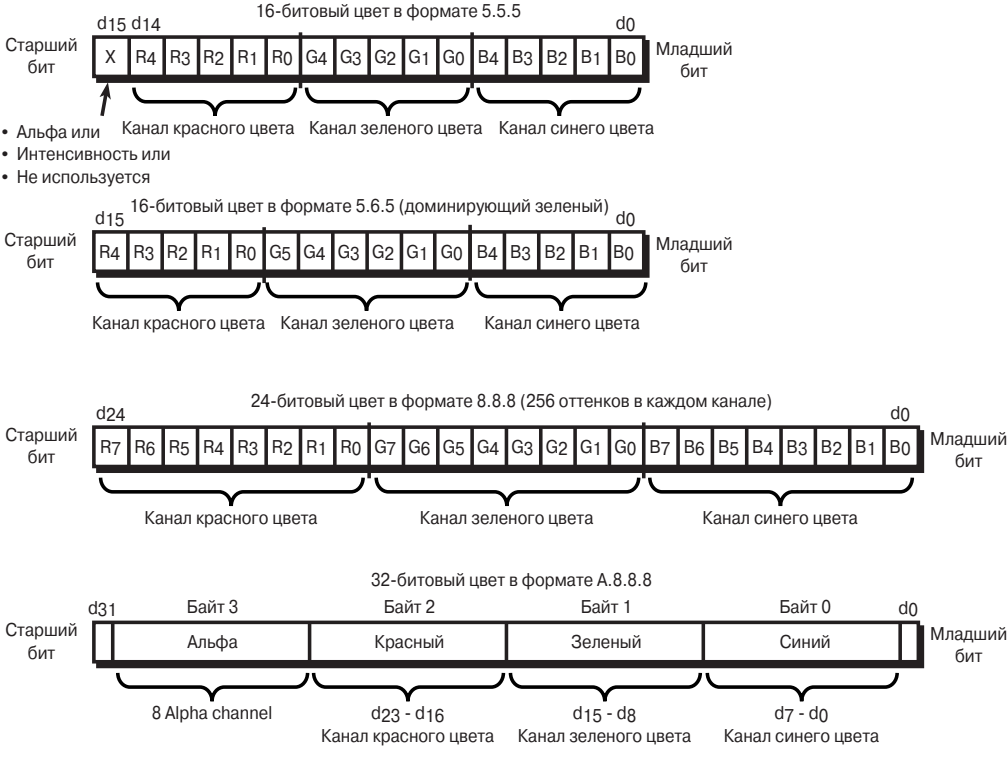
Видеорежимы RGB и палитры

Имеется два способа представления цвета — прямой и косвенный. Прямые видеорежимы, или режимы RGB, представляют каждый пиксель на экране с помощью 16, 24 или 32 бит, представляющих красную, зеленую и синюю составляющие цвета на экране (рис. 3.15). Это возможно в силу аддитивной природы основных цветов — красного, синего и зеленого. Поскольку некоторые числа целиком на 3 не делятся, составляющие цвета могут быть представлены разным количеством бит. Например, 16-битовые цвета могут быть закодированы следующим образом:

- RGB (6.5.5) — 6 бит для представления красного цвета и по 5 бит для зеленого и синего;
- RGB (1.5.5.5) — 1 бит для альфа-канала и по 5 бит для представления красного, зеленого и синего цветов (альфа-канал управляет прозрачностью);
- RGB (5.6.5) — 6 бит для представления зеленого цвета и по 5 бит для красного и синего. Мой опыт подсказывает, что это наиболее распространенный способ кодирования.

В 24-битовом режиме каждому каналу почти всегда выделяется по 8 бит, но в 32-битовом режиме в большинстве случаев происходит то же, а 8 оставшихся бит выделяются для альфа-канала.

RGB-режимы предоставляют возможность точного управления соотношением красного, зеленого и синего компонентов каждого пикселя на экране. Использование же палитры можно просто считать дополнительным уровнем косвенности. При использовании только 8 бит на пиксель выделение 2–3 бит на цвет не позволяет получить большое число оттенков каждого цвета, поэтому в таком режиме используется палитра.



Примечание: некоторые видеокарты поддерживают формат 10.10.10

Рис. 3.15. Кодирование цвета в RGB-режимах

Как показано на рис. 3.16, *палитра* представляет собой таблицу с 256 записями, по одной для каждого возможного значения одного байта — от 0 до 255. Каждая запись состоит из трех 8-битовых значений для красного, зеленого и синего цветов, по сути представляя собой 24-битовый RGB дескриптор. *Таблица поиска цветов* (Color lookup table — CLUT) работает следующим образом. При чтении из видеопамяти значение очередного пикселя, например 26, используется в качестве индекса в таблице поиска цветов; затем считанное из таблицы 24-битовое значение — для получения выводимого на экран цвета пикселя. Таким путем мы можем использовать на экране одновременно 256 цветов из 16,7 миллионов.

Конечно, рассматривая эту тему, мы несколько забегаем вперед, но я хотел дать вам общие представления о данных концепциях с тем, чтобы при рассмотрении DirectDraw вы уже имели определенные знания в этом вопросе. В действительности цвет — настолько серьезная проблема, что Windows использует абстрактную 24-битовую модель цвета, которая позволяет вам не беспокоиться о глубине цвета и подобных вещах при программировании. Конечно, если вы побеспокоитесь об этом, результаты могут оказаться значительно привлекательнее, но делать это вы не обязаны.

Вывод текста

Система вывода текста в Windows — одна из наиболее сложных и надежных из когда-либо виденных мною. Конечно, большинству программистов игр достаточно только выводить счет, тем не менее познакомиться с этой системой не помешает.

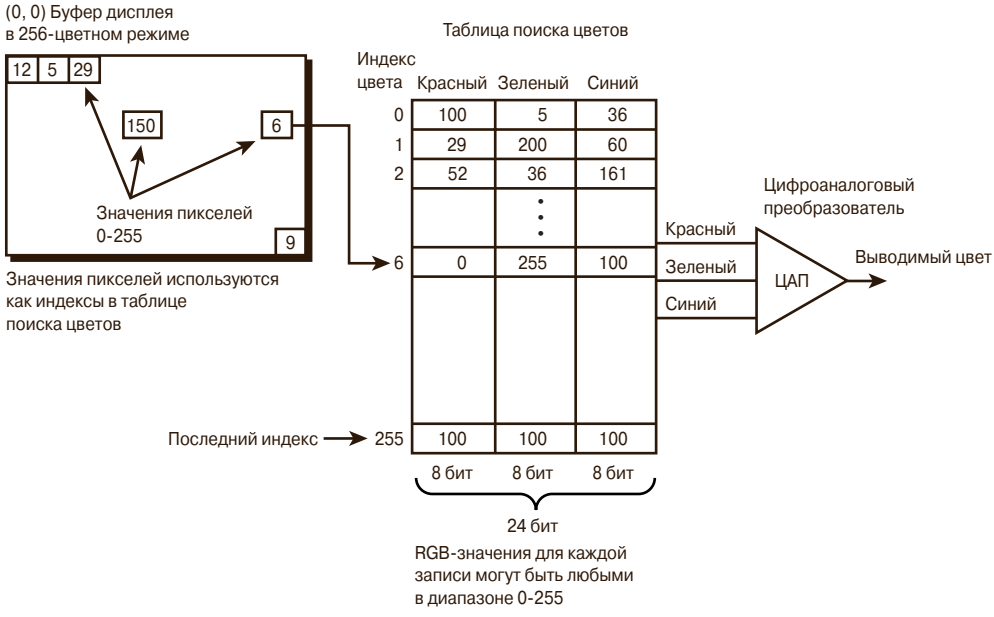


Рис. 3.16. Принцип работы палитры в 8-битовом режиме

В действительности вывод текста с использованием GDI обычно слишком медленный для применения в играх, так что в конечном итоге нам понадобится создание собственного средства текстового вывода на базе DirectX. Но пока что мы познакомимся с выводом текста с использованием GDI. В крайнем случае это пригодится нам при отладке и в демоверсиях.

Для вывода текста используются две популярные функции — TextOut() и DrawText(). Я обычно обращаюсь к первой — она побыстрее, да и я не использую всех этих "рюшечек и финтифлюшечек", предоставляемых второй функцией. Впрочем, мы все равно познакомимся с обеими. Итак, вот их прототипы:

```
BOOL TextOut(HDC hdc,  // Дескриптор контекста устройства
     int    nxStart,   // x-координата начала вывода
     int    nyStart,   // y-координата начала вывода
     LPCTSTR lpString, // Адрес выводимой строки
     int    cbString); // Количество символов в строке

int DrawText(HDC hdc,  // Дескриптор контекста устройства
     LPCTSTR lpString, // Адрес выводимой строки
     int    nCount,    // Количество символов в строке
     LPRECT  lpRect,   // Указатель на ограничивающий
                       // прямоугольник
     UINT   uFormat);  // Флаги вывода текста
```

Назначение большинства параметров очевидно. При использовании TextOut() вы передаете функции контекст устройства, координаты начала вывода x и y и строку вместе с ее длиной. Функция DrawText() немного сложнее. Поскольку она осуществляет форматирование и размещение текста, ей в качестве параметра передается прямоугольник, в котором этот текст следует разместить (рис. 3.17), и флаги, указывающие, как именно это следует сделать (например, определяющие выравнивание текста). Познакомиться со списком флагов можно в справочной системе Win32 SDK, а я упомяну только об одном из них — DT_LEFT, который выравнивает весь текст по левой границе.

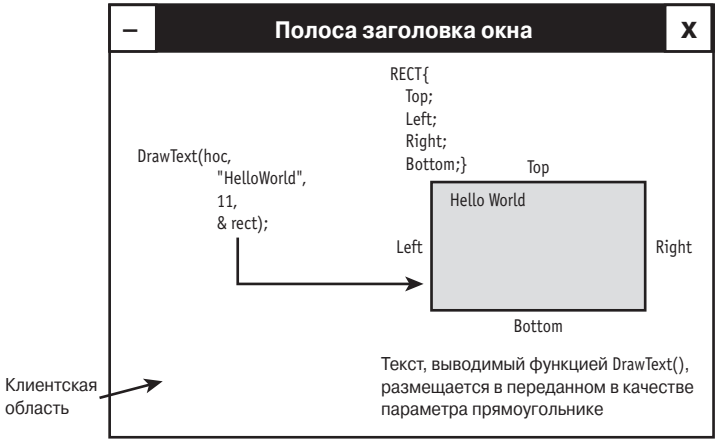


Рис. 3.17. Вывод текста функцией DrawText()

Единственная проблема — в обеих функциях ничего не упоминается о цвете. Не правда ли, несколько странно? Дело в том, что, к счастью, имеется способ указать как цвет вывода текста, так и цвет фона, а также прозрачность вывода.

Режим прозрачности текста указывает, каким образом должны выводиться символы текста. Должны ли они выводиться с фоновым прямоугольником или следует прорисовывать только пиксели текста? На рис. 3.18 показаны разные варианты вывода текста. Как видите, в режиме прозрачности текст выглядит как нарисованный поверх графики, что гораздо привлекательнее, чем вывод текста с фоновым прямоугольником.

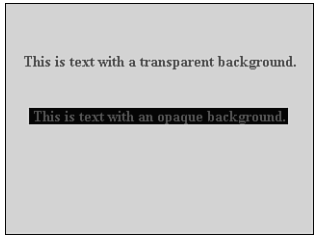


Рис. 3.18. Разные режимы вывода текста

СОВЕТ

Непрозрачный вывод текста осуществляется быстрее, так что текст на одноцветном фоне вы можете просто выводить с соответствующим фоновым цветом.

Вот функции, которые определяют основной и фоновый цвета текста:

```
COLORREF SetTextColor( HDC hdc,  // Дескриптор контекста
            COLORREF color);     // Цвет текста
COLORREF SetBkColor  ( HDC hdc,  // Дескриптор контекста
            COLORREF color);     // Цвет фона
```

Обе функции получают в качестве параметра контекст графического устройства (полученный вызовом GetDC() или BeginPaint()) и цвет в формате COLORREF. После того как вы определили цвет, он будет использоваться до тех пор, пока вы не измените его с помощью вызова той же функции. При изменении цвета каждая функция возвращает цвет, который использовался для вывода текста до этого изменения, так что вы можете сохранить его и восстановить после завершения работы.

Что? Вы уже готовы выводить разноцветный текст, но не знаете, что такое формат COLORREF? Это очень просто!

```
typedef struct tagCOLORREF
{
  BYTE bRed;      // Красная составляющая
  BYTE bGreen;    // Зеленая составляющая
  BYTE bBlue;     // Синяя составляющая
  BYTE bDummy;    // Не используется
} COLORREF;
```

Таким образом, в памяти COLORREF выглядит как 0x00bbggrr. (Не забывайте, что в архитектуре PC первым идет младший байт.) Для получения корректного значения COLORREF можно использовать макрос RGB().

```
COLORREF red    = RGB(255, 0, 0);
COLORREF yellow = RGB(255,255, 0);
```

И так далее. Если мы рассмотрим структуру дескриптора цвета, то обнаружим, что она выглядит практически так же, как и структура PALETTEENTRY.

```
typedef struct tagPALETTEENTRY
{
  BYTE peRed;    // Красная составляющая
  BYTE peGreen;  // Зеленая составляющая
  BYTE peBlue;   // Синяя составляющая
  BYTE peFlags;  // Управляющие флаги
} PALETTEENTRY;
```

Поле peFlags может принимать значения, показанные в табл. 3.3. В большинстве случаев используются флаги PC_NOCOLLAPSE и PC_RESERVED, но пока что вам достаточно просто знать, что они существуют. Я всего лишь хотел показать аналогичность PALETTEENTRY и COLORREF (за исключением интерпретации последнего байта), означающую, что во многих случаях они взаимозаменяемы.

Таблица 3.3. Флаги PALETTEENTRY

Значение	*Описание*
PC_EXPLICIT	Указывает, что младшее слово записи логической палитры определяет индекс аппаратной палитры
PC_NOCOLLAPSE	Указывает, что цвет размещен в неиспользуемой записи системной палитры, а не соответствует одному из имеющихся в ней цветов
PC_RESERVED	Указывает, что запись логической палитры используется для анимации палитры. Этот флаг предотвращает сравнение цвета окном с цветом в данной записи, поскольку он часто изменяется. Если в системной палитре имеются неиспользуемые записи, цвет помещается в одну из них; в противном случае цвет для анимации недоступен

Теперь вы уже почти готовы к выводу текста. Почему почти? Потому что осталось рассмотреть, как же осуществить вывод текста с прозрачным фоном. Для этого можно воспользоваться следующей функцией:

```
int SetBkMode( HDC hdc,      // Дескриптор контекста
       int iBkMode);         // Режим прозрачности
```

Эта функция наряду с дескриптором контекста устройства получает указывающий режим прозрачности параметр, который может принимать значения TRANSPARENT или OPAQUE. Функция возвращает старое значение режима прозрачности, которое вы можете сохранить и впоследствии восстановить.

Итак, вы уже почти совсем готовы к выводу текста. Почему почти совсем? Потому что осталось только написать код этого вывода. Вот как вы можете вывести текст в окно.

```
COLORREF old_fcolor,    // Старый цвет текста
      old_bcolor;       // Старый цвет фона

int    old_tmode;       // Старый режим прозрачности

// Получаем контекст графического устройства
HDC hdc = GetDC(hwnd);

// Устанавливаем цвета и режим вывода текста
// и сохраняем старые значения

old_fcolor = SetTextColor(hdc, RGB(0,255,0));
old_bcolor = SetBkColor(hdc, RGB(0,0,0));
old_tmode = SetBkMode(hdc, TRANSPARENT);

// Выводим текст в позиции (20, 30)
TextOut(hds, 20, 30, "Hello", strlen("Hello"));

// Восстанавливаем старые параметры вывода
SetTextColor(hdc, old_fcolor);
SetBkColor(hdc, old_bcolor);
SetBkMode(hdc, old_tmode);

// Освобождаем контекст устройства
ReleaseDC(hwnd, hdc);
```

Разумеется, нет никакого закона, по которому вас можно привлечь к ответственности за невосстановленные старые значения. Я просто показал, как это делается, тем более что эти параметры действительны только для данного контекста устройства и при освобождении последнего теряют смысл. Однако представим ситуацию, когда вы хотите вывести один текст зеленым цветом, а второй синим. В этом случае вам достаточно сменить цвет, не закрывая контекст устройства и не используя все три функции, определяющие цвет и режим вывода текста.

В качестве примера вывода текста с использованием описанной технологии рассмотрите демонстрационную программу DEMO3_5.CPP на прилагаемом компакт-диске. Эта программа заполняет окно расположенными случайным образом текстовыми строками различных цветов (рис. 3.19).

Рис. 3.19. Вывод текста программой DEMO3_5.EXE

Приведем фрагмент кода из функции WinMain() данного приложения, где происходит вывод текста.

```
// Получаем и сохраняем контекст устройства
HDC hdc = GetDC(hwnd);

// Главный цикл событий с использованием PeekMessage()
// для выборки сообщений из очереди
while(TRUE)
  {
  // Проверка наличия сообщения и выборка
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // Проверка необходимости выхода
    if (msg.message == WM_QUIT)
      break;

    // Трансляция быстрых клавиш
    TranslateMessage(&msg);

    // Обработка сообщения
    DispatchMessage(&msg);
    } // if

  // Работа игры

  // Выбор случайного цвета
  SetTextColor(hdc, RGB(rand()%256,
          rand()%256,rand()%256));

  // Черный фон
  SetBkColor(hdc, RGB(0,0,0));

  // Вывод текста с прозрачным фоном
  SetBkMode(hdc, TRANSPARENT);

  // Вывод текста в случайной позиции
  TextOut(hdc,rand()%400,rand()%400, "GDI Text Demo!",
       strlen("GDI Text Demo!"));

  Sleep(10);

  } // while

// Освобождение контекста устройства
ReleaseDC(hwnd,hdc);
```

В качестве второго примера вывода текста попробуем выводить значение счетчика, фиксирующего, сколько раз было послано сообщение WM_PAINT. Вот соответствующий код:

```
char      buffer[80];                 // Строка вывода
static int wm_paint_count = 0;  // Счетчик сообщений

.
.
.
```

```
case WM_PAINT:
{
   hdc = BeginPaint(hwnd,&ps);

   // Синий цвет текста
   SetTextColor(hdc, RGB(0,0,255));
   // Черный цвет фона
   SetBkColor(hdc, RGB(0,0,0));
   // Непрозрачный вывод
   SetBkMode(hdc, OPAQUE);

   // Вывод текста со значением счетчика
   sprintf(buffer,"WM_PAINT called %d times.   ",
        ++wm_paint_count);
   TextOut(hdc, 0,0, buffer, strlen(buffer));
   EndPaint(hwnd,&ps);

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

Полный текст программы вы найдете в файле DEMO3_6.CPP на прилагаемом компакт-диске. Запустите программу DEMO3_6.EXE и посмотрите на нее в действии. Обратите внимание на то, что до тех пор, пока вы не измените размер окна или не перекроете его другим, значение счетчика не изменяется, так как сообщение WM_PAINT не посылается[1].

На этом мы завершаем рассмотрение вывода текста. Конечно, функция DrawText() умеет гораздо больше, а использование других функций позволяет использовать различные шрифты для вывода текста. Но в любом случае это не для нас, так как наша цель — написание игр, а не какого-то текста...

Обработка основных событий

Как уже много (пожалуй, слишком много!) раз упоминалось, Windows представляет собой операционную систему, базирующуюся на событиях. Одним из наиболее важных аспектов стандартной программы Windows является реакция на события. В этом разделе мы рассмотрим только некоторые наиболее важные события, обработки которых вам будет вполне достаточно при написании игр.

Управление окном

Существует ряд сообщений, которые Windows отправляет в качестве уведомления о том, что пользователь манипулирует вашим окном. Список некоторых наиболее интересных сообщений этого класса приведен в табл. 3.4.

[1] В качестве "домашнего задания" попытайтесь перекрыть это окно другим так, чтобы выводимая надпись была не перекрыта. Затем несколько раз переместите перекрывающее окно, не затрагивая при этом выведенный текст; вы увидите, что выведенная строка при этом не изменится, но после того, как вы перекроете выведенный текст, произойдет резкое увеличение выводимого значения — больше, чем на 1. Объясните этот эффект и измените программу так, чтобы в окне всегда отображалось актуальное значение счетчика. Подсказка: вспомните разницу между вызовом функций GetDC() и BeginPaint(). — *Прим. ред.*

Таблица 3.4. Сообщения управления окном

Значение	*Описание*
WM_ACTIVATE	Посылается, когда окно активизируется или деактивизируется. Это сообщение сначала отправляется процедуре деактивизируемого окна верхнего уровня, а затем — процедуре активизируемого окна верхнего уровня
WM_ACTIVATEAPP	Посылается, когда активизируется окно, принадлежащее другому приложению. Сообщение посылается как приложению, окно которого становится активным, так и приложению, окно которого перестает быть активным
WM_CLOSE	Посылается как сигнал о том, что окно или приложение должно завершить свою работу
WM_MOVE	Посылается после перемещения окна
WM_MOVING	Посылается окну, перемещаемому пользователем. При обработке этого сообщения приложение может отслеживать размер и положение перемещаемого прямоугольника и при необходимости изменять его размер или положение
WM_SIZE	Посылается окну после изменения его размера
WM_SIZING	Посылается окну, размер которого изменяется пользователем. При обработке этого сообщения приложение может отслеживать размер и положение перемещаемого прямоугольника и при необходимости изменять его размер или положение

Рассмотрим сообщения WM_ACTIVATE, WM_CLOSE, WM_SIZE и WM_MOVE более подробно.

Сообщение WM_ACTIVATE

Параметры:

```
fActive    = LOWORD(wparam);          // Флаг активизации
fMinimized = (BOOL)HIWORD(wparam); // Флаг минимизации
hwndPrevious = (HWND)lparam;          // Дескриптор окна
```

Параметр fActive определяет, что произошло с окном, т.е. стало ли оно активным или, наоборот, неактивным? Эта информация хранится в младшем слове параметра wparam и может принимать одно из значений, показанных в табл. 3.5.

Таблица 3.5. Флаг активизации сообщения WM_ACTIVATE

Значение	*Описание*
WA_CLICKACTIVE	Активизируется щелчком мыши
WA_ACTIVE	Окно активизируется каким-то иным способом, например при помощи клавиатуры
WA_INACTIVE	Окно деактивизируется

Флаг fMinimized просто показывает, находится ли окно в минимизированном состоянии (это так, если значение флага отлично от нуля). И наконец, hwndPrevious указывает окно, которое будет активизировано или деактивизировано, в зависимости от значения флага fActive. Если это значение равно WA_INACTIVE, то hwndPrevious указывает окно, которое будет активизировано; в противном случае этот параметр указывает окно, которое станет неактивным. Данный дескриптор может также принимать значениеNULL.

Мы используем это сообщение, если хотим знать о том, что наше приложение стало активным или, напротив, перестало быть таковым.

Вот набросок кода, обрабатывающего это сообщение:

```
case WM_ACTIVATE:
{
  // Проверка состояния окна
  if (LOWORD(wparam) != WA_INACTIVE)
  {
    // Приложение активизируется
  }
  else
  {
    // Приложение деактивизируется
  }
} break;
```

Сообщение WM_CLOSE

Параметры: отсутствуют.

Данное сообщение отправляется перед тем, как послать сообщение WM_DESTROY с последующим WM_QUIT. Это сообщение указывает, что пользователь пытается закрыть окно вашего приложения. Если ваш обработчик вернет нулевое значение, ничего не произойдет; таким образом, пользователь не сможет закрыть ваше окно. Посмотрите исходный текст приложения DEMO3_7.CPP, а затем запустите приложение DEMO3_7.EXE и попробуйте закрыть его окно!

ВНИМАНИЕ

Не паникуйте, если не можете закрыть окно приложения DEMO3_7.EXE. Нажмите <Ctrl+Alt+Del>, и перед вами появится окно диспетчера задач, который справится с непокорным приложением.

Вот как выглядит фрагмент кода обработчика данного сообщения в программе DEMO3_7.CPP:

```
case WM_CLOSE:
{
  // Убиваем сообщение, так что сообщение
  // WM_DESTROY послано не будет
  return(0);
} break;
```

Если ваша цель — свести пользователя с ума, то можете воспользоваться приведенным фрагментом. Однако лучше использовать обработку этого сообщения для того, чтобы вывести диалоговое окно, запрашивающее подтверждение закрытия приложения, а заодно выполнить все необходимые действия по освобождению захваченных ресурсов и т.п. Логика такого решения представлена на рис. 3.20.

Далее приведен код обработчика сообщения WM_CLOSE из демонстрационной программы DEMO3_8.CPP.

```
case WM_CLOSE:
{
  // Вывод диалогового окна
  int result =  MessageBox(hwnd,
    "Are you sure you want to close this application?",
    "WM_CLOSE Message Processor",
    MB_YESNO | MB_ICONQUESTION);

  // Пользователь подтверждает выход?
  if (result == IDYES)
```

```
  {
    // Вызов обработчика по умолчанию
    return (DefWindowProc(hwnd, msg, wparam, lparam));
  }
  else // Игнорируем сообщение
  return(0);

} break;
```

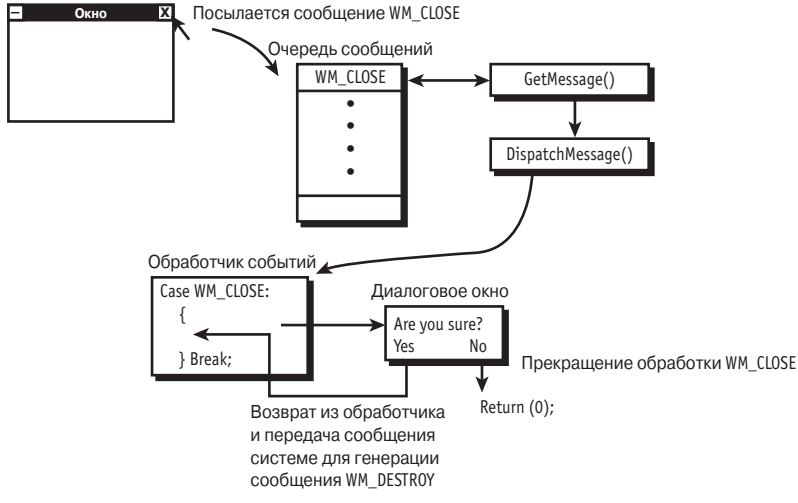


Рис. 3.20. Интеллектуальный обработчик сообщения WM_CLOSE

Обратите внимание на вызов обработчика по умолчанию DefWindowProc(). Этот вызов осуществляется, когда пользователь подтверждает необходимость закрытия окна. Обработчик по умолчанию посылает сообщение WM_DESTROY, которое могли бы послать и вы сами. Так как мы пока не научились отправлять сообщения, воспользуемся обработчиком по умолчанию.

Сообщение WM_SIZE

Параметры:

```
fwSizeType = wparam;          // Флаг изменения размера
nWidth     = LOWORD(lparam); // Ширина клиентской области
nHeight    = HIWORD(lparam); // Высота клиентской области
```

Параметр fwSizeType указывает, какое именно изменение размеров окна произошло (табл. 3.6), а младшее и старшее слова параметра lparam указывают новые размеры окна.

Таблица 3.6. Параметр fwSizeType сообщения WM_SIZE

Значение	*Описание*
SIZE_MAXHIDE	Сообщение посылается всем всплывающим окнам, когда некоторое другое окно максимизируется
SIZE_MAXIMIZED	Окно максимизировано
SIZE_MAXSHOW	Сообщение посылается всем всплывающим окнам, когда некоторое другое окно возвращается к своему предыдущему размеру

Значение	*Описание*
SIZE_MINIMIZED	Окно минимизировано
SIZE_RESTORED	Окно изменило размеры, но при этом не применимы ни значение SIZE_MAXIMIZED, ни SIZE_MINIMIZED

Это сообщение очень важно при написании игр, поскольку при изменении размера окна графика должна быть масштабирована таким образом, чтобы полностью занимать окно. Этого не может произойти, если ваша игра использует полноэкранную графику, но в случае оконной игры пользователь может захотеть изменить размеры окна. В этом случае вы должны принять меры к тому, чтобы изображение в окне оставалось корректным.

В демонстрационной программе DEMO3_9.CPP обработчик данного сообщения отслеживает новый размер при изменении размеров окна пользователем и выводит его в текстовой строке в окне.

```
case WM_SIZE:
   {
   // Получение информации о новом размере окна
   int width  = LOWORD(lparam);
   int height = HIWORD(lparam);

   // Получение контекста устройства
   hdc = GetDC(hwnd);

   // Установка цвета и режима вывода строки
   SetTextColor(hdc, RGB(0,255,0));
   SetBkColor(hdc, RGB(0,0,0));
   SetBkMode(hdc, OPAQUE);

   // Вывод размера окна
   sprintf(buffer,"WM_SIZE Called -  New Size = (%d,%d)",
         width, height);
   TextOut(hdc, 0,0, buffer, strlen(buffer));

   // Освобождение контекста устройства
   ReleaseDC(hwnd, hdc);

   } break;
```

ВНИМАНИЕ

Вы должны знать об одной потенциальной проблеме, связанной с обработкой сообщения WM_SIZE. Дело в том, что при изменении размеров окна посылается не только сообщение WM_SIZE, но и сообщение WM_PAINT. Соответственно, если сообщение WM_PAINT посылается после сообщения WM_SIZE, то код обработки сообщения WM_PAINT может обновить фон окна и тем самым стереть информацию, выведенную при обработке сообщения WM_SIZE. К счастью, в нашей ситуации этого не происходит, но это хороший пример проблем, возникающих при изменении порядка сообщений или когда они посылаются не в том порядке, который вы ожидаете.

Очень похоже на WM_SIZE сообщение WM_MOVE. Основное отличие заключается в том, что это сообщение посылается не при изменении размеров окна, а при его перемещении.

Сообщение WM_MOVE

Параметры:

```
// Новое положение окна по горизонтали в экранных координатах
xPos = (int)LOWORD(lparam);

// Новое положение окна по вертикали в экранных координатах
yPos = (int)HIWORD(lparam);
```

Сообщение WM_MOVE посылается, когда окно оказывается перемещенным в новое положение, как показано на рис. 3.21. Следует обратить ваше внимание на то, что сообщение посылается *после* перемещения, а не во время его в реальном времени (если вы хотите отслеживать сам процесс перемещения, то для этого имеется сообщение WM_MOVING, но в большинстве случаев на время перемещения окна пользователем вся работа приостанавливается).

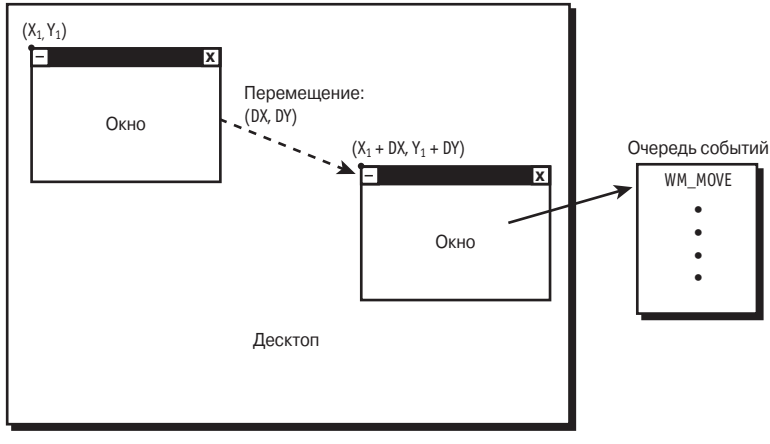


Рис. 3.21. Генерация сообщения MW_MOVE

Примером отслеживания движения окна может служить демонстрационная программа DEMO3_10.CPP, обработка сообщения WM_MOVE в которой приводит к выводу в окне его новых координат.

```
case WM_MOVE:
   {
   // Получение координат окна
   int xpos = LOWORD(lparam);
   int ypos = HIWORD(lparam);

   // Получение контекста устройства
   hdc = GetDC(hwnd);

   // Установка цвета и режима вывода строки
   SetTextColor(hdc, RGB(0,255,0));
   SetBkColor(hdc, RGB(0,0,0));
   SetBkMode(hdc, OPAQUE);

   // Вывод координат окна
   sprintf(buffer,"WM_MOVE Called -  New Positition "
         "= (%d,%d)", xpos, ypos);
   TextOut(hdc, 0,0, buffer, strlen(buffer));
```

```
// Освобождение контекста устройства
ReleaseDC(hwnd, hdc);

} break;
```

На этом мы остановимся. Сообщений, связанных с управлением окнами, очень много, но мы просто не можем позволить себе рассматривать их все. Если когда-нибудь вам понадобится отследить какие-то манипуляции с окном, обратитесь к справочной системе Win32 и будьте уверены: вы непременно найдете сообщение, которое решит вашу задачу.

Теперь же мы перейдем к устройствам ввода, которые обеспечивают взаимодействие программы с пользователем (и делают демонстрационные программы гораздо интереснее).

Работа с клавиатурой

Когда-то работа с клавиатурой требовала определенного шаманства. Вы должны были писать обработчик прерываний, создавать таблицу состояний и делать еще кучу других вещей в надежде, что все это заработает. Я сам — низкоуровневый программист, но без всякого сожаления должен сказать, что мне удавалось увильнуть от этой работы.

В конечном счете для работы с клавиатурой, мышью, джойстиком и другими устройствами ввода мы будем использовать возможности DirectInput, тем не менее вы должны иметь представление о том, как использовать для доступа к клавиатуре и мыши библиотеку Win32. Если даже это и не пригодится ни для чего иного, кроме написания демонстрационных программ, пока мы не доберемся до DirectInput.

Клавиатура состоит из множества клавиш, микроконтроллера и электроники, поддерживающей работу клавиатуры. При нажатии клавиш(и) в Windows посылается последовательный поток пакетов, описывающих нажатые клавиши(у). Затем Windows обрабатывает этот поток и посылает вам сообщение о событии клавиатуры. При этом получить доступ к сообщениям клавиатуры можно разными способами.

- Посредством сообщения `WM_CHAR`.
- Посредством сообщений `WM_KEYDOWN` и `WM_KEYUP`.
- При помощи вызова `GetAsyncKeyState()`.

Каждый из этих способов работает немного по-разному. Сообщения `WM_CHAR` и `WM_KEYDOWN` генерируются Windows при нажатии клавиши, однако передаваемая в этих сообщениях информация различна. При нажатии клавиши на клавиатуре, например <A>, генерируется следующая информация:

- скан-код;
- код символа.

Скан-код представляет собой уникальный код, назначенный каждой клавишей клавиатуры и не имеющий ничего общего с вводимым символом. Во многих случаях вам просто надо знать, не была ли нажата клавиша <A>, и вас не интересует, например, была ли в этот момент нажата клавиша <Shift> или нет. По сути, вы хотите использовать клавиатуру как множество моментальных переключателей, что и делается с использованием скан-кодов и сообщения `WM_KEYDOWN`, которое генерируется при нажатии клавиши.

Код символа — это уже "приготовленные" данные. Это означает, что если вы нажали клавишу <A> и при этом не была нажата клавиша <Shift> и не включен режим <Caps Lock>, то вы получите символ `'a'`. Аналогично, при нажатии комбинации клавиш <Shift+A> вы получите символ `'A'`. Эта информация передается посредством сообщения `WM_CHAR`.

Вы можете использовать любой метод — тот, который больше подходит для ваших задач. Если вы работаете над текстовым редактором, вы, вероятно, будете использовать сообщение WM_CHAR, поскольку в этом случае вас интересует вводимый символ, а не скан-коды клавиш. С другой стороны, если в вашей игре <F> служит для стрельбы, <S> для удара, а <Shift> для обороны, то как реагировать на введенные символы? Вам просто нужно знать, какие клавиши на клавиатуре нажаты.

Последний метод состоит в чтении состояния клавиатуры с помощью функции Win32 GetAsyncKeyState(), которая получает информацию о последнем известном состоянии клавиатуры в виде таблицы состояний. Я предпочитаю именно этот метод, так как при его использовании не требуется написание обработчика клавиатуры.

Теперь, когда мы вкратце узнали о каждом методе работы с клавиатурой, познакомимся с ними поближе, начав с сообщения WM_CHAR.

Сообщение WM_CHAR содержит следующую информацию:

wparam — код символа;

lparam — битовый вектор состояния, описывающий различные управляющие клавиши, которые также могут оказаться нажатыми (табл. 3.7).

Таблица 3.7. Вектор состояния клавиатуры

Биты	*Описание*
0 – 15	Содержит *счетчик повторений*, представляющий собой количество повторно введенных символов в результате удерживания клавиши в нажатом состоянии
16 – 23	Содержит скан-код. Это значение зависит от производителя аппаратного обеспечения (OEM)
24	Логическое значение, указывающее, является ли нажатая клавиша расширенной (например, правой <Alt> или <Ctrl>)
29	Логическое значение, указывающее, нажата ли клавиша <Alt>
30	Логическое значение, указывающее предыдущее состояние клавиши
31	Логическое значение, указывающее состояние клавиши. Если это значение равно 1, клавиша отпущена; в противном случае — нажата

Для обработки сообщения WM_CHAR все, что вы должны сделать, — это написать его обработчик, который выглядит примерно так:

```
case WM_CHAR
{
  // Получение информации о клавиатуре
  int ascii_code = wparam;
  int key_state  = lparam;

  // Действия, вызванные клавиатурой

} break;
```

Разумеется, вы можете проверять интересующее вас состояние клавиатуры. Например, вот как проверить, нажата или нет клавиша <Alt>:

```
// Проверка состояния 29-го бита key_state
#define ALT_STATE_BIT 0x20000000
if (key_state & ALT_STATE_BIT)
{
```

```
  // Выполнение соответствующих действий
}  // if
```

Аналогично вы можете проверить и другие состояния клавиатуры.

В качестве примера обработки сообщения WM_CHAR рассмотрите демонстрационную программу DEMO3_11.CPP, в которой при нажатии клавиши осуществляется вывод информации о символе и векторе состояния клавиатуры.

```
case WM_CHAR:
{
  // Получение информации
  char ascii_code = wparam;
  unsigned int key_state = lparam;

  // Получение контекста устройства
  hdc = GetDC(hwnd);

  // Установка цвета и режима вывода строки
  SetTextColor(hdc, RGB(0,255,0));
  SetBkColor(hdc, RGB(0,0,0));
  SetBkMode(hdc, OPAQUE);

  // Вывод кода символа и состояния клавиатуры
  sprintf(buffer,"WM_CHAR: Character = %c   ",ascii_code);
  TextOut(hdc, 0,0, buffer, strlen(buffer));

  sprintf(buffer,"Key State = 0X%X  ",key_state);
  TextOut(hdc, 0,16, buffer, strlen(buffer));

  // Освобождение контекста устройства
  ReleaseDC(hwnd, hdc);

} break;
```

Следующее сообщение клавиатуры, WM_KEYDOWN, похоже на WM_CHAR, с тем отличием, что вся информация поступает в "необработанном" виде. Данные представляют собой виртуальный скан-код, который отличается от стандартного скан-кода тем, что гарантируется его независимость от типа используемой клавиатуры.

wparam — виртуальный код нажатой клавиши. В табл. 3.8 приведены некоторые наиболее часто употребляемые виртуальные коды.

lparam — битовый вектор состояния, описывающий различные управляющие клавиши, которые также могут оказаться нажатыми (см. табл. 3.7).

Таблица 3.8. Виртуальные коды клавиш

Символьное имя	*Шестнадцатеричное значение*	*Описание*
VK_BACK	08	Backspace
VK_TAB	09	Tab
VK_RETURN	0D	Enter
VK_SHIFT	10	Shift
VK_CONTROL	11	Ctrl

Окончание табл. 3.8

Символьное имя	*Шестнадцатеричное значение*	*Описание*
VK_PAUSE	13	Pause
VK_ESCAPE	1B	Esc
VK_SPACE	20	Spacebar
VK_PRIOR	21	PgUp
VK_NEXT	22	PgDn
VK_END	23	End
VK_HOME	24	Home
VK_LEFT	25	←
VK_UP	26	↑
VK_RIGHT	27	→
VK_DOWN	28	↓
VK_INSERT	2D	Insert
VK_DELETE	2E	Delete
VK_HELP	2F	Help
Отсутствует	30 — 39	0 — 9
Отсутствует	41 — 5A	A — Z
VK_F1 – VK_F12	70 — 7B	F1 — F12

Примечание. Для клавиш A–Z и 0–9 символьные имена отсутствуют. Вы можете использовать для работы с ними числовые значения или определить собственные символьные имена.

В дополнение к сообщению WM_KEYDOWN имеется сообщение WM_KEYUP. Оно имеет те же параметры и отличается от WM_KEYDOWN только тем, что генерируется при отпускании клавиши.

Вот набросок кода для обработки сообщения WM_KEYDOWN:

```
case WM_KEYDOWN:
{
  // Получение виртуального кода и вектора состояния
  int virtual_code = (int)wparam;
  int key_state    = (int)lparam;

  // Выполнение действий в зависимости от кода
  switch(virtual_code)
  {
    case VK_RIGHT: {} break;
    case VK_LEFT : {} break;
    case VK_UP   : {} break;
    case VK_DOWN : {} break;

    // Прочие коды

    default: break;
  } // switch
```

```
    // Сообщаем Windows о том, что сообщение обработано
    return(0);
} break;
```

В качестве эксперимента замените обработчик сообщений WM_CHAR в демонстрационной программе DEMO3_11.CPP обработчиком сообщения WM_KEYDOWN и посмотрите, что из этого получится.

Последний метод чтения клавиатуры состоит в вызове одной из функций опроса состояния клавиатуры: GetKeyboardState(), GetKeyState() или GetAsyncKeyState(). Мы рассмотрим последнюю из них, поскольку она работает для одной клавиши, что обычно интересует нас гораздо больше, чем состояние всей клавиатуры. Если вас заинтересуют остальные функции, обратитесь за информацией к справочной системеWin32 SDK.

Функция GetAsyncKeyState() имеет следующий прототип:

```
SHORT GetAsyncKeyState(int virtual_key);
```

Вы просто передаете функции виртуальный код клавиши, состояние которой хотите протестировать, и, если старший бит возвращаемого значения равен 1, эта клавиша нажата; в противном случае она находится в свободном состоянии. Обычно я использую для проверки пару макросов, которые облегчают написание соответствующего кода.

```
#define KEYDOWN(vk_code) \
        ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEYUP(vk_code)   \
        ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)
```

Преимущество использования функции GetAsyncKeyState() в том, что она не привязана к циклу событий. Вы можете проверить нажатие клавиши в любой момент. Например, вы создаете игру и хотите отслеживать клавиши перемещения курсора, пробел и <Ctrl>. Для этого вам не нужно дожидаться события WM_CHAR или WM_KEYDOWN — вы просто пишете код наподобие следующего:

```
if (KEYDOWN(VK_DOWN))
{
  // Обработка нажатой клавиши↓
} // if
if (KEYDOWN(VK_UP))
{
  // Обработка нажатой клавиши↑
} // if

// И так далее...
```

Аналогично проверяется, что некоторая клавиша не нажата:

```
if (KEYUP(VK_ENTER))
{
  // Соответствующие действия
} // if
```

В качестве примера использования функции GetAsyncKeyState() я разработал демонстрационную программу DEMO3_12.CPP, код функции WinMain() которой приведен ниже.

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

```
{

WNDCLASSEX winclass;
HWND     hwnd;
MSG      msg;
HDC      hdc;

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

// Сохранение hinstance в глобальной переменной
hinstance_app = hinstance;

// Регистрация класса
if (!RegisterClassEx(&winclass))
   return(0);

// Создание окна
if (!(hwnd = CreateWindowEx(NULL,
                    WINDOW_CLASS_NAME,
                    "GetAsyncKeyState() Demo",
                    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                    0,0,
                    400,300,
                    NULL,
                    NULL,
                    hinstance,
                    NULL)))
return(0);

// Сохранение дескриптора окна
main_window_handle = hwnd;

// Вход в главный цикл (использующий PeekMessage())
while(TRUE)
   {
   // Проверка наличия сообщений
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
     {
     if (msg.message == WM_QUIT)
        break;

     TranslateMessage(&msg);
```

```
    // Передача сообщения процедуре окна
    DispatchMessage(&msg);
    } // if

    // Работа игры

    // Получение контекста устройства
    hdc = GetDC(hwnd);

    // Установка цвета и режима вывода строки
    SetTextColor(hdc, RGB(0,255,0));
    SetBkColor(hdc, RGB(0,0,0));
    SetBkMode(hdc, OPAQUE);

    // Вывод информации о состоянии
    // клавиш перемещения курсора
    sprintf(buffer,"Up Arrow: = %d",KEYDOWN(VK_UP));
    TextOut(hdc, 0,0, buffer, strlen(buffer));

    sprintf(buffer,"Down Arrow: = %d",KEYDOWN(VK_DOWN));
    TextOut(hdc, 0,16, buffer, strlen(buffer));

    sprintf(buffer,"Right Arrow: = %d",KEYDOWN(VK_RIGHT));
    TextOut(hdc, 0,32, buffer, strlen(buffer));

    sprintf(buffer,"Left Arrow: = %d",KEYDOWN(VK_LEFT));
    TextOut(hdc, 0,48, buffer, strlen(buffer));

    // Освобождение контекста устройства
    ReleaseDC(hwnd, hdc);

  } // while

// Возврат в Windows
return(msg.wparam);

} // WinMain
```

Если вы просмотрите полный код программы DEMO3_12.CPP на прилагаемом компакт-диске, то увидите, что в нем нет обработчиков сообщений WM_CHAR или WM_KEYDOWN. Чем меньше сообщений обрабатывается процедурой окна, тем лучше. Кроме того, это первая программа, в которой в части кода, относящейся к работе игры, выполняются некоторые действия. Пока что здесь отсутствует какая-либо синхронизация и работа со временем, так что перерисовка экрана идет с максимально возможной скоростью. В главе 4, "GDI, управляющие элементы и прочее", вы познакомитесь с соответствующими технологиями, которые позволят обновлять экран с заранее заданной частотой кадров. А пока что мы поближе познакомимся с работой мыши.

Работа с мышью

Мышь, пожалуй, наиболее удобное средство ввода. Вы просто указываете нужное место на экране и щелкаете — всего лишь! Для работы с мышью Windows предлагает два класса сообщений: WM_MOUSEMOVE и WM_*BUTTON*.

Начнем с сообщения WM_MOUSEMOVE. Прежде всего следует запомнить, что координаты мыши указываются относительно клиентской области окна, в котором она находится.

Взгляните на рис. 3.22: координаты указателя мыши отсчитываются от верхнего левого угла вашего окна, координаты которого равны (0, 0).

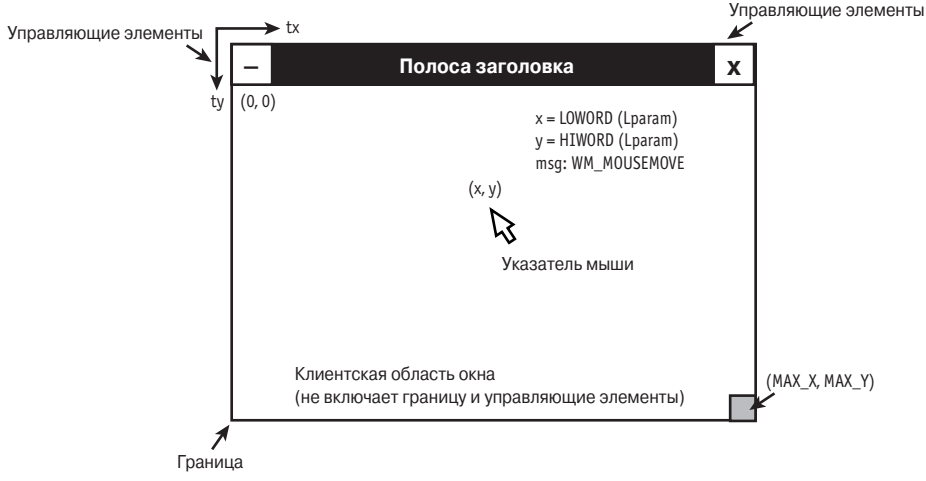


Рис. 3.22. Подробности перемещения мыши

Сообщение WM_MOUSEMOVE

Параметры:

```
int mouse_x = (int)LOWORD(lparam);
int mouse_y = (int)HIWORD(lparam);

int buttons = (int)wparam;
```

Позиция указателя кодируется как две 16-битовых записи в параметре lparam, а состояние кнопок мыши описывается битовыми значениями параметра wparam, представленными в табл. 3.9.

Таблица 3.9. Состояние кнопок мыши

Значение	*Описание*
MK_LBUTTON	Бит установлен, если нажата левая кнопка мыши
MK_MBUTTON	Бит установлен, если нажата средняя кнопка мыши
MK_RBUTTON	Бит установлен, если нажата правая кнопка мыши
MK_CONTROL	Бит установлен, если нажата клавиша <Ctrl>
MK_SHIFT	Бит установлен, если нажата клавиша <Shift>

Таким образом, для проверки нажатия той или иной кнопки мыши вы можете использовать битовую операцию AND с соответствующим значением из табл. 3.9. Вот пример кода, отслеживающего положение мыши вместе с состояниями ее левой и правой кнопок:

```
case WM_MOUSEMOVE:
{
  // Получение положения указателя мыши
  int mouse_x = (int)LOWORD(lparam);
  int mouse_y = (int)HIWORD(lparam);
```

```
  // Получение состояния кнопок мыши
  int buttons = (int)wparam;

  // Проверка нажатия левой кнопки
  if (buttons & MK_LBUTTON)
  {
    // Соответствующие действия
  } // if

  // Проверка нажатия правой кнопки
  if (buttons & MK_RBUTTON)
  {
    // Соответствующие действия
  } // if

} break;
```

Не правда ли, тривиально? В качестве демонстрационного примера отслеживания положения мыши взгляните на программу DEMO3_13.CPP на прилагаемом к книге компакт-диске. Программа выводит положение мыши и состояние ее кнопок. Обратите внимание, что данная программа замечает изменения состояния кнопок только при перемещении указателя мыши (чего и следовало ожидать, так как сообщение посылается при перемещении мыши, а не при нажатии ее кнопок).

Операционная система не гарантирует постоянное отправление сообщения WM_MOUSEMOVE — вы, например, можете просто перемещать мышь слишком быстро. Таким образом, нельзя полагаться на то, что вы сможете хорошо отследить отдельные перемещения мыши. Обычно из-за этого не возникает никаких трудностей, тем не менее помнить об этом надо. А чтобы следить за изменениями состояния кнопок мыши, когда она не двигается, существует целый ряд сообщений, перечисленных в табл. 3.10.

Таблица 3.10. Сообщения от кнопок мыши

Сообщение	*Описание*
WM_LBUTTONDBLCLK	Двойной щелчок левой кнопкой мыши
WM_LBUTTONDOWN	Левая кнопка мыши нажата
WM_LBUTTONUP	Левая кнопка мыши отпущена
WM_MBUTTONDBLCLK	Двойной щелчок средней кнопкой мыши
WM_LBUTTONDOWN	Средняя кнопка мыши нажата
WM_LBUTTONUP	Средняя кнопка мыши отпущена
WM_RBUTTONDBLCLK	Двойной щелчок правой кнопкой мыши
WM_RBUTTONDOWN	Правая кнопка мыши нажата
WM_RBUTTONUP	Правая кнопка мыши отпущена

Эти сообщения от кнопок мыши также содержат координаты указателя мыши, как и в сообщении WM_MOUSEMOVE. Таким образом, для отслеживания двойных щелчков левой кнопкой мыши можно использовать приведенный далее код.

```
case WM_LBUTTONDBLCLK:
{
  // Получение положения указателя мыши
  int mouse_x = (int)LOWORD(lparam);
  int mouse_y = (int)HIWORD(lparam);

  // Некоторые действия в ответ на двойной щелчок

  // Сообщаем Windows, что сообщение обработано
  return(0);

} break;
```

Отправление сообщений самому себе

Последнее, о чем я хотел рассказать в этой главе, как отправлять сообщения самому себе. Для этого есть два пути.

- SendMessage(). Отправляет сообщение окну для немедленной обработки. Возврат из функции осуществляется только после того, как процедура получающего сообщение окна обработает данное сообщение.
- PostMessage(). Отправляет сообщение в очередь окна и тут же возвращает управление. Используется, если вас не беспокоит задержка перед обработкой посылаемого сообщения или если ваше сообщение имеет невысокий приоритет.

Прототипы обеих функций очень похожи:

```
LRESULT
  SendMessage(HWND  hWnd, // Дескриптор окна-получателя
        UINT  Msg,        // Отправляемое сообщение
        WPARAM wparam,    // Первый параметр сообщения
        LPARAM lparam);   // Второй параметр сообщения

BOOL
  PostMessage(HWND  hWnd,  // Дескриптор окна-получателя
        UINT  Msg,         // Отправляемое сообщение
        WPARAM wparam,     // Первый параметр сообщения
        LPARAM lparam);    // Второй параметр сообщения
```

Возвращаемое функцией SendMessage() значение представляет собой значение, которое возвращает процедура окна-получателя. Функция PostMessage() при успешном завершении возвращает ненулевое значение. Как видите, возвращаемые значения этих функций различны. Почему? Потому что SendMessage() вызывает процедуру окна, в то время как PostMessage() просто помещает сообщение в очередь сообщений окна-получателя без какой-либо обработки.

И все-таки зачем посылать сообщение самому себе? Тому есть буквально миллионы причин. Не забывайте, что Windows — операционная система, управляемая сообщениями, и все в мире Windows должно работать единообразно, путем отправления и обработки сообщений. В следующей главе вы познакомитесь с управляющими элементами типа кнопок и узнаете, что единственный способ работы с ними состоит в отправлении им сообщений. Ну если вы не верите мне, то поверьте конкретным примерам.

Работу всех демонстрационных программ мы завершали с помощью управляющего элемента закрытия окна или комбинации клавиш <Alt+F4>. Но как добиться того же эффекта программно?

Вы знаете, что работа окна завершается отправлением ему сообщений WM_CLOSE или WM_DESTROY (первое из них, как вы помните, позволяет выполнить определенные действия и даже отменить закрытие окна). Таким образом, чтобы решить нашу задачу, можно просто послать окну сообщение WM_DESTROY вызовом

```
SendMessage(hwnd,WM_DESTROY,0,0);
```

или, если вас не смущает небольшая задержка, вызовом

```
PostMessage(hwnd,WM_DESTROY,0,0);
```

В обоих случаях приложение прекращает работу, если, конечно, вы не изменили логику обработки сообщения WM_DESTROY. Далее возникает вопрос, а когда, собственно, следует посылать такое сообщение? Это зависит от вас. Например, если игра прекращается нажатием клавиши <Esc>, то в цикле событий можно воспользоваться ранее описанным макросом KEYDOWN.

```
if (KEYDOWN_VK_ESCAPE))
  SendMessage(hwnd,WM_CLOSE,0,0);
```

Чтобы посмотреть работу этого кода, обратитесь к демонстрационной программе DEMO3_14.CPP на прилагаемом компакт-диске. В качестве эксперимента попробуйте изменить сообщение на WM_DESTROY и воспользоваться функцией PostMessage().

ВНИМАНИЕ

Отправление сообщений вне главного цикла событий может привести к непредвиденным проблемам. Например, в описанной только что ситуации вы закрываете окно извне основного цикла событий, отправляя сообщение непосредственно процедуре окна вызовом SendMessage(). Таким образом можно вызвать *ошибку нарушения порядка выполнения*, состоящую в том, что некоторое событие наступает раньше другого, в то время как обычно их порядок обратный. Не забывайте о возможности нарушения порядка выполнения при отправке сообщений самому себе и учтите, что функция PostMessage() в этом плане более безопасна, поскольку работает через очередь сообщений.

И наконец, вы можете послать себе собственное пользовательское сообщение WM_USER. Это сообщение отправляется с помощью любой из рассмотренных функций; параметры wparam и lparam при этом могут иметь тот смысл (и значения), который вы в них вложите. Например, вы можете использовать это сообщение, чтобы создать ряд виртуальных сообщений для системы управления памятью.

```
// Определения для диспетчера памяти
#define ALLOC_MEM   0
#define DEALLOC_MEM 1

// Отправляем сообщение WM_USER, используя lparam
// для передачи количества памяти, а wparam для
// указания типа запрашиваемой операции
SendMessage(hwnd,WM_USER,ALLOC_MEM,1000);
```

Посланное сообщение обрабатывается в процедуре окна.

```
case WM_USER:
{
  // Какое именно виртуальное сообщение получено?
  switch(wparam)
  {
    case ALLOC_MEM:  {} break;
```

```
      case DEALLOC_MEM: {} break;
      // ... прочие сообщения ...
    } // switch
} break;
```

Как видите, в параметрах wparam и lparam вы можете закодировать любую требующуюся вам информацию и поступить с ней так, как сочтете нужным.

Резюме

Наконец-то! Мне уже начало казаться, что я никогда не закончу эту главу: ресурсы, меню, GDI, сообщения — не слишком ли много материала для одной главы? Хороший учебник по программированию в Windows должен иметь объем не менее 3000 страниц, так что вы видите всю сложность моей задачи — свести количество страниц до 58. Надеюсь, мне удалось отобрать действительно важный материал и при всей сжатости объяснить его достаточно доходчиво. Думаю, прочитав следующую главу, вы будете знать о Windows все необходимое для того, чтобы начать работу над играми.

ГЛАВА 4

GDI, управляющие элементы и прочее

Перед вами последняя глава, посвященная программированию в Windows. В ней более детально рассматривается использование GDI, в частности вывод пикселей, линий и простых фигур. Затем затрагиваются вопросы синхронизации, и завершается глава описанием управляющих элементов Windows. И наконец, рассматривается создание шаблона приложения T3D Game Console, который послужит отправной точкой для всех демонстрационных приложений в оставшейся части книги.

Работа с GDI

Как уже отмечалось, GDI работает существенно медленнее, чем DirectX. Тем не менее не стоит отказываться от использования (и изучения) GDI. Более того, вполне возможно совместное использование GDI и DirectX.

Здесь вы познакомитесь только с основными операциями GDI. Дополнительную информацию о GDI вы можете почерпнуть из справочной системы Win32 SDK; полученные в этой главе основы помогут вам быстро и легко разобраться во всех вопросах, связанных с GDI.

Контекст графического устройства

В главе 3, "Профессиональное программирование в Windows", вы неоднократно встречались с типом, который представляет собой дескриптор контекста устройства — `HDC`. В нашем случае контекст устройства является контекстом графического устройства, но имеются и другие контексты устройств, например контекст принтера. В любом случае вы можете заинтересоваться: а что же в действительности представляет собой контекст устройства, что на самом деле означает это понятие?

Контекст графического устройства, по сути, представляет собой описание видеокарты, установленной в вашей системе. Следовательно, когда вы обращаетесь к контексту графического устройства или дескриптору, это означает, что, в конечном счете, вы обращаетесь к описанию видеокарты, ее разрешений и возможностей работы с цветом. Эта информация требуется для любого графического вызова GDI. Дескриптор контекста устройства, передаваемый в каждую функцию GDI, используется в качестве ссылки на важную информацию о видеосистеме, с которой работают данные функции.

Кроме того, контекст графического устройства отслеживает программные установки, которые могут измениться в процессе работы программы. Например, GDI использует ряд графических объектов — *перья*, *кисти*, *стили линий* и т.п. Описания этих базовых данных используются GDI для вывода любых графических примитивов, которые могут вам понадобиться. Таким образом, изменение, например, текущего цвета пера (который не является свойством видеокарты) будет отслежено контекстом графического устройства. Таким образом, контекст графического устройства представляет собой не только описание аппаратного обеспечения видеосистемы, но и хранилище информации о ваших настройках, так что при вызове GDI вам не требуется явно передавать их в вызываемую функцию. А теперь, помня о том, что же такое контекст графического устройства, рассмотрим вывод графики с помощью GDI.

Цвет, перья и кисти

Если подумать, то найдется не так уж много типов объектов, которые можно изобразить на экране компьютера. Да, конечно, количество различных фигур почти не ограничено, но количество типов объектов при этом весьма невелико. Это *точки*, *линии* и *многоугольники*. Все, что выводится на экран, представляет собой комбинацию этих типов примитивных объектов.

Подход, используемый GDI, напоминает работу художника, который создает картины с помощью красок, перьев и кистей; точно так же поступает и GDI с помощью своих инструментов.

- **Перья** используются для вывода линий или контуров. Они имеют толщину, цвет и стиль.
- **Кисти** применяются для заполнения замкнутых объектов. Они имеют цвет, стиль и могут даже представлять собой растровые изображения (рис. 4.1).

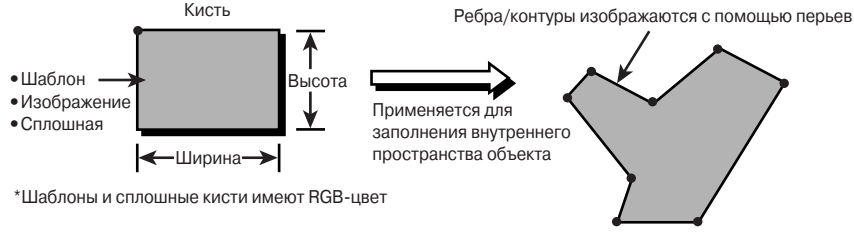


Рис. 4.1. Использование кисти

Перед тем как вы возьмете в руки перья и кисть и начнете творить, я бы хотел сделать еще одно замечание. GDI одновременно использует только одно перо и одну кисть. Конечно, в вашем распоряжении может быть множество перьев и кистей, но в текущем контексте графического устройства активными могут быть только одно перо и одна кисть. Это означает, что вы должны "выбрать объект", перед тем как его использовать.

Вспомните, что контекст графического устройства — это не только описание видеокарты и ее возможностей, но и описание текущего инструментария. Перья и кисти пред-

ставляют собой первые примеры инструментов, которые отслеживаются контекстом графического устройства, и вы должны выбирать необходимый инструмент для работы с данным контекстом графического устройства. Этот процесс называется "*выбором*". Главное — запомните, что выбранный в данном контексте графического устройства инструмент будет использоваться до тех пор, пока не будет заменен другим.

И последнее: если вы создаете перо или кисть, вы должны их удалить. Это важное требование, поскольку Windows GDI имеет ограниченное количество слотов для дескрипторов перьев и кистей и вы можете легко исчерпать их.

Работа с перьями

Дескриптор пера имеет тип HPEN. Вот как создается нулевое перо:

```
HPEN pen_1 = NULL;
```

Здесь pen_1 представляет собой дескриптор пера, но самого пера нет — оно не создано и его свойства не определены. Это делается одним из двух способов.

- Использование готового объекта.
- Создание пользовательского пера.

Использование готового объекта из семейства объектов (stock objects) того или иного типа основано на том, что Windows имеет ряд готовых к использованию объектов с предопределенными свойствами. Вам нужно просто выбрать готовый объект из семейства с помощью функции GetStockObject(), которая возвращает дескрипторы различных объектов, включая перья, кисти и шрифты.

```
HGDIOBJ GetStockObject(int fnObject); // Тип готового объекта
```

Эта функция просто выбирает готовый объект и возвращает вам его дескриптор. Типы готовых перьев показаны в табл. 4.1.

Таблица 4.1. Типы готовых объектов

Значение	*Описание*
BLACK_PEN	Черное перо
NULL_PEN	Нулевое перо
WHITE_PEN	Белое перо
BLACK_BRUSH	Черная кисть
DKGRAY_BRUSH	Темно-серая кисть
GRAY_BRUSH	Серая кисть
HOLLOW_BRUSH	Пустая кисть (эквивалент NULL_BRUSH)
LTGRAY_BRUSH	Светло-серая кисть
NULL_BRUSH	Нулевая кисть (эквивалент HOLLOW_BRUSH)
WHITE_BRUSH	Белая кисть
ANSI_FIXED_FONT	Стандартный моноширинный системный шрифт Windows
ANSI_VAR_FONT	Стандартный пропорциональный системный шрифт Windows
DEFAULT_GUI_FONT	Только для Windows 95: шрифт по умолчанию для объектов пользовательского интерфейса, таких, как меню и диалоговые окна

Значение	*Описание*
OEM_FIXED_FONT	Моноширинный шрифт производителя аппаратного обеспечения
SYSTEM_FONT	Системный шрифт. По умолчанию этот шрифт используется Windows при выводе меню, управляющих элементов диалоговых окон и текста. В Windows 3.0 и более поздних используется пропорциональный шрифт; до этого использовался моноширинный
SYSTEM_FIXED_FONT	Моноширинный системный шрифт, использовавшийся в версиях Windows до 3.0. Этот объект предоставляется для совместимости с ранними версиями Windows

Как видно из табл. 4.1, в GDI не так уж много готовых перьев... Но раз уж мы заговорили о них, то создать, например, белое перо можно следующим образом:

```
HPEN white_pen = GetStockObject(WHITE_PEN);
```

Разумеется, GDI пока что ничего не знает об этом пере, поскольку оно не было выбрано в данном контексте графического устройства, но об этом чуть позже.

Более интересный метод — самостоятельное создание перьев с определением их цвета, стиля и ширины. Функция, которая создает перья, имеет следующий прототип:

```
HPEN CreatePen(int     fnPenStyle, // Стиль пера
       int     nWidth,             // Ширина пера
       COOLORRFE crColor);         // Цвет пера
```

Параметры nWidth и crColor вполне очевидны, а о параметре fnPenStyle следует сказать несколько слов дополнительно.

В большинстве случаев вы, вероятно, захотите рисовать сплошные линии, но иногда требуется вывод пунктирных линий. Конечно, можно попробовать изобразить множество отрезков с небольшими промежутками между ними, но не проще ли поручить эту работу GDI? В табл. 4.2 приведены различные стили линий, которые вы можете использовать при работе с GDI.

Таблица 4.2. Стили линий, используемые функцией CreatePen()

Стиль	*Описание*
PS_NULL	Невидимая линия
PS_SOLID	Сплошная линия
PS_DASH	Пунктирная линия
PS_DOT	Линия, состоящая из точек
PS_DASHDOT	Линия, состоящая из чередующихся точек и отрезков
PS_DASHDOTDOT	Линия, состоящая из чередующихся отрезков и двойных точек

В качестве примера создадим три пера, рисующих сплошные линии разного цвета толщиной 1 пиксель.

```
// Красное перо
HPEN red_pen   = CreatePen(PS_SOLID,1,RGB(255,0,0));
// Зеленое перо
HPEN green_pen = CreatePen(PS_SOLID,1,RGB(0,255,0));
// Синее перо
HPEN blue_pen  = CreatePen(PS_SOLID,1,RGB(0,0,255));
```

А вот как создается перо для вычерчивания пунктирных линий:

```
HPEN white_dashed_pen =
        CreatePen(PS_DASHED,1,RGB(255,255,255))
```

Все очень просто, не правда ли? А теперь давайте рассмотрим, как пояснить контексту графического устройства, с каким именно пером ему предстоит работать. Для выбора любого объекта GDI в контекст графического устройства используется функция

```
HGDIOBJ SelectObject(
    HDC hdc,              // Дескриптор контекста устройства
    HGDIOBJ hgdiobj); // Дескриптор объекта
```

Функция SelectObject() получает в качестве параметров дескриптор контекста графического устройства и дескриптор выбираемого объекта. Заметим, что функция *полиморфна*, т.е. она может получать дескрипторы объектов разных типов. Причина в том, что все дескрипторы графических объектов являются подклассами типа данных HGDIOBJ (дескриптор объекта GDI). Функция возвращает дескриптор текущего объекта, который вы заменяете новым (т.е., выбирая новое перо, вы можете получить дескриптор старого, которое перестает быть выбранным, сохранить это значение и позже восстановить его). Вот пример выбора нового пера и сохранения старого:

```
HDC hdc;  // Контекст графического устройства (предполагается,
          // что он имеет корректное значение)

// Создание синего пера
HPEN blue_pen = CreatePen(PS_SOLID,1,RGB(0,0,255));

HPEN old_pen = NULL; // Переменная для хранения старого пера

// Выбор синего пера и сохранение старого пера
old_pen = SelectObject(hdc, blue_pen);

// Черчение линий новым пером

// Восстановление старого пера
SelectObject(hdc, old_pen);
```

И в конце, когда вы завершите работу с созданными перьями (с помощью как функции CreatePen(), так и GetStockObject()), вы должны уничтожить их посредством вызова функции DeleteObject(), которая также является полиморфной и может удалять объекты разного типа. Вот ее прототип:

```
BOOL DeleteObject(
    HGDIOBJ hObject); // Дескриптор графического объекта
```

ВНИМАНИЕ

Будьте внимательны при уничтожении перьев. Если вы удаляете выбранный в настоящее время объект или выбираете уже уничтоженный объект, то это приведет к ошибке, а возможно, к сбою защиты памяти.

НА ЗАМЕТКУ

В приводимых фрагментах кода и демонстрационных программах я не уделяю должного внимания проверке ошибок, но совершенно очевидно, что в реальных программах вы должны проверять возвращаемые функциями значения, чтобы убедиться в их успешности. В противном случае вы можете столкнуться с неприятностями.

Когда же именно следует вызывать функцию DeleteObject()? Обычно это делается в конце программы, но, если вы создаете сотни объектов, используете их и больше до конца программы они вам не нужны, можете тут же удалить их. Дело в том, что ресурсы Windows GDI далеко не безграничны. Вот как мы можем уничтожить все созданные нами в приведенных ранее примерах перья:

```
DeleteObject(red_pen);
DeleteObject(green_pen);
DeleteObject(blue_pen);
DeleteObject(white_dashed_pen);
```

ВНИМАНИЕ

Не пытайтесь повторно удалить уже удаленные объекты. Это может привести к непредсказуемым результатам.

Работа с кистями

Теперь рассмотрим, что собой представляют кисти. В действительности они во всем подобны перьям, за исключением внешнего вида. Кисти используются для закраски графических объектов, в то время как перья предназначаются для рисования контуров объектов или простых линий. Однако и к тем и к другим применимы одни и те же принципы. Дескриптор кисти имеет тип HBRUSH, а пустой объект кисти создается присвоением:

```
HBRUSH brush_1 = NULL;
```

Для того чтобы кисть стала действительной кистью, которой можно рисовать, вы можете либо выбрать готовую кисть с помощью вызова GetStockObject() (см. табл. 4.1), либо определить собственную. Вот как, например, создать светло-серую кисть:

```
brush_1 = GetStockObject(LTGRAY_BRUSH);
```

Для создания более интересных кистей вы можете выбрать тип шаблона заполнения и цвет — так же, как и для перьев. К сожалению, GDI разбивает кисти на два класса: сплошные и штрихованные. Лично мне это разделение представляется не очень удачным: на мой взгляд, GDI должен рассматривать все кисти как штрихованные, а сплошную кисть — как частный случай штриховки. Но что поделаешь! Итак, функция для создания сплошной кисти называется CreateSolidBrush() и имеет следующий прототип:

```
HBRUSH CreateSolidBrush(COLORREF crColor);  // Цвет кисти
```

Для создания зеленой сплошной кисти вы должны выполнить следующее:

```
HBRUSH green_brush = CreateSolidBrush(RGB(0,255,0));
```

А для выбора этой кисти в контекст графического устройства:

```
HBRUSH old_brush = NULL;

old_brush = SelectObject(hdc, green_brush);

// Использование кисти для рисования

// Восстанавливаем старую кисть
SelectObject(hdc, old_brush);
```

В конце программы вы должны удалить объект кисти:

```
DeleteObject(green_brush);
```

В общем, все как обычно: создаем объект, выбираем, пользуемся им и удаляем.

Теперь посмотрим, как создать штрихованную кисть. Для этого вы должны использовать показанную ниже функцию CreateHatchBrush().

```
HBRUSH CreateHatchBrush(int fnStyle,     // Стиль штриховки
        COLORREF clrref); // Цвет кисти
```

Значение стиля штриховки может быть одним из представленных в табл. 4.3.

Таблица 4.3. Возможные значения стиля штриховки

Значение	*Описание*
HS_BDIAGONAL	Штриховка слева направо вниз под углом 45°
HS_CROSS	Горизонтальные и вертикальные линии
HS_DIAGCROSS	Диагональные линии в двух направлениях
HS_FDIAGONAL	Штриховка слева направо вверх под углом 45°
HS_HORIZONTAL	Горизонтальная штриховка
HS_VERTICAL	Вертикальная штриховка

В качестве последнего примера, связанного с кистями, создадим красную кисть с вертикальной штриховкой и используем ее:

```
HBRUSH red_brush = CreateHatchBrush(HS_VERTICAL,RGB(255,0,0));
HBRUSH old_brush = SelectObject(hdc, red_brush);

// Использование кисти

SelectObject(hdc, old_brush);
DeleteObject(red_hbrush);
```

Точки, линии, многоугольники и окружности

Теперь, когда вы познакомились с концепциями перьев и кистей, самое время узнать, как их использовать в реальных программах для рисования различных объектов. Начнем изучение с простейшего графического объекта.

Точка

Вывод точки в GDI тривиален и не требует ни пера, ни кисти (очевидно, что, поскольку точка представляет собой единственный пиксель, ни перья, ни кисти к нему не применимы). Для вывода точки в клиентской области окна вам нужны контекст графического устройства окна, координаты и цвет точки:

```
COLORREF SetPixel(HDC hdc,   // Контекст устройства
        int x,               // x-координата
        int y,               // y-координата
        COLORREF crColor); // Цвет пикселя
```

Эта функция, получающая в качестве параметров контекст графического устройства окна, координаты и цвет выводимой точки, выводит пиксель в указанной точке и возвращает цвет, который на самом деле получила данная точка. Дело в том, что, например, если в 256-цветном режиме указанный вами RGB-цвет не существует, GDI использует ближайший к нему и вернет его вам.

Если вы не уверены в том, что означают координаты точки (x, y), взгляните на рис. 4.2. На нем изображена оконная система координат, использующаяся GDI, которая представляет собой инвертированную декартову систему координат: координата x увеличивается слева направо, а y растет сверху вниз.

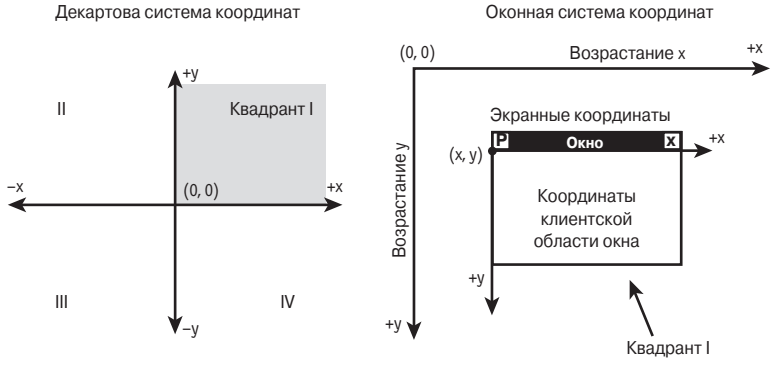


Рис. 4.2. Декартова и оконная системы координат

Технически GDI может использовать и другие координаты, но данная система координат принята по умолчанию и используется как GDI, так и DirectX. Обратите внимание, что точка (0,0) представляет собой верхний левый угол клиентской области окна, а не всего окна. Вы можете получить контекст графического устройства всего окна, воспользовавшись функцией GetWindowDC() вместо GetDC(). При использовании полученного таким образом контекста графического устройства вы можете осуществлять вывод в любом месте окна, включая его управляющие элементы. Вот пример вывода 1000 точек случайного цвета со случайным положением в окне, размер которого известен заранее и равен 400×400:

```
HWND hwnd; // Предполагается, что здесь хранится
           // корректный дескриптор окна
HDC  hdc;

// Получаем контекст графического устройства
hdc = GetDC(hwnd);

for(int index = 0; index < 1000; ++index)
{
  // Получаем случайные координаты и цвет
  int x = rand()%400;
  int y = rand()%400;
  COLORREF color = RGB(rand()%255, rand()%255,
                       rand()%255);

  SetPixel(hdc,x,y,color);
} // for index
```

В качестве примера рассмотрите демонстрационную программу DEMO1_4.CPP, в которой приведенный фрагмент кода выполняется в бесконечном цикле. Внешний вид работающего приложения показан на рис. 4.3. (В качестве “домашнего задания” попробуйте исправить программу так, чтобы заполнение точками осуществлялось во всем окне даже при увеличении его размеров. После этого замените вызов функции GetDC() вызовом GetWindowDC() и посмотрите, что получится в результате. — *Прим. ред.*)

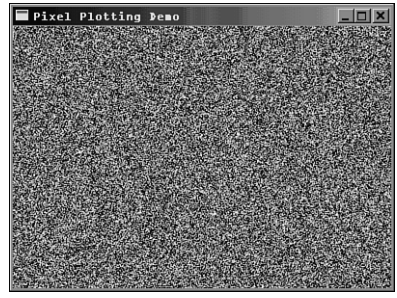


Рис. 4.3. Демонстрационная программа DEMO4_1.EXE

Линия

Следующий после точки по сложности примитив — линия. Для того чтобы прочертить линию, нужно создать перо и вызвать функцию черчения линии в соответствии с приведенным алгоритмом.

1. Создайте перо и выберите его. Все линии будет чертиться с использованием этого пера.
2. Установите начальные координаты линии.
3. Прочертите линию из начальных координат в конечные (которые представляют собой начальные координаты следующего отрезка).
4. Перейдите к п. 3 и начертите столько отрезков, сколько требуется.

По сути, GDI имеет невидимый курсор, который отслеживает текущую начальную позицию вычерчиваемой линии. Первоначально вами должна быть указана эта позиция, но затем при вычерчивании линии GDI автоматически обновляет ее, облегчая тем самым черчение сложных объектов типа многоугольников. Вот функция, которая перемещает начальный курсор в нужную позицию:

```
BOOL MoveToEx(
  HDC hdc,              // Дескриптор контекста устройства
  int x,                // x-координата новой позиции
  int y,                // y-координата новой позиции
  LPPOINT lpPoint);  // Адрес для хранения старой позиции
```

Предположим, вы хотите провести линию от точки (10,10) к точке (50,60). В таком случае вы должны начать с вызова наподобие этого:

```
// Устанавливаем текущую позицию курсора
MoveToEx(hdc, 10, 10, NULL);
```

Обратите внимание, что значение последнего параметра функции — NULL. Если вы хотите сохранить предыдущее положение курсора, делайте так:

```
POINT last_pos;

// Устанавливаем текущую позицию курсора
MoveToEx(hdc, 10, 10, &last_pos);
```

Напомню, как выглядит структура POINT:

```
typedef struct tagPOINT
{
  LONG x;
  LONG y;
} POINT;
```

После того как начальное положение курсора определено, вычертить отрезок можно при помощи функции

```
BOOL LineTo(HDC hdc,   // Дескриптор контекста устройства
      int xEnd,        // Конечная x-координата
      int yEnd);       // Конечная y-координата
```

Итак, полностью фрагмент кода для вывода зеленой сплошной линии от точки (10, 10) к точке (50, 60) выглядит следующим образом:

```
HWND  hwnd;  // Предполагается, что здесь хранится
             // корректный дескриптор окна

// Получаем контекст графического устройства
HDC   hdc = GetDC(hwnd);

// Создаем зеленое перо
HPEN green_pen = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));

// Выбираем перо в контекст графического устройства
HPEN old_pen = SelectObject(hdc, green_pen);

// Чертим линию
MoveToEx(hdc, 10, 10, NULL);
LineTo(hdc, 50, 60);

// Восстанавливаем старое перо
SelectObject(hdc, old_pen);

// Удаляем зеленое перо
DeleteObject(green_pen);

// Освобождаем контекст устройства
ReleaseDC(hwnd, hdc);
```

Если вы хотите нарисовать треугольник с вершинами (20, 10), (30, 20), (10, 20), то вот как должен выглядеть соответствующий код:

```
// Начальная вершина
MoveToEx(hdc, 20, 10, NULL);

// Черчение треугольника
LineTo(hdc, 30, 20);
LineTo(hdc, 10, 20);
LineTo(hdc, 20, 10);
```

Теперь вы видите, в чем преимущество использования пары функций MoveToEx() — LineTo()?

В качестве примера черчения линий рассмотрите демонстрационную программу DEMO4_2.CPP, которая выводит случайные линии в окне приложения. На рис. 4.4 вы можете увидеть эту программу в действии.

Прямоугольник

Следующим объектом, который мы рассмотрим, будет прямоугольник. При рисовании прямоугольника используются и перо и кисть (при заполнении внутренней области прямоугольника). Для этого вызывается функция

```
BOOL Rectangle(HDC hdc,  // Дескриптор контекста устройства
    int nLeftRect,       // Левая граница прямоугольника
    int nTopRect,        // Верхняя граница прямоугольника
    int nRightRect,      // Правая граница прямоугольника
    int nBottomRect);    // Нижняя граница прямоугольника
```

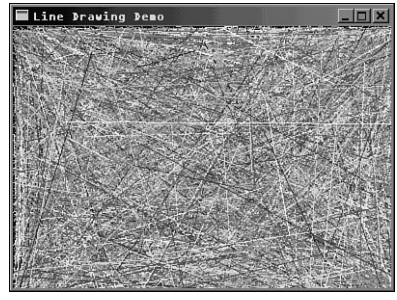


Рис. 4.4. Демонстрационная программа DEMO4_2.EXE

Функция Rectangle() рисует прямоугольник с использованием текущих пера и кисти, как показано на рис. 4.5.

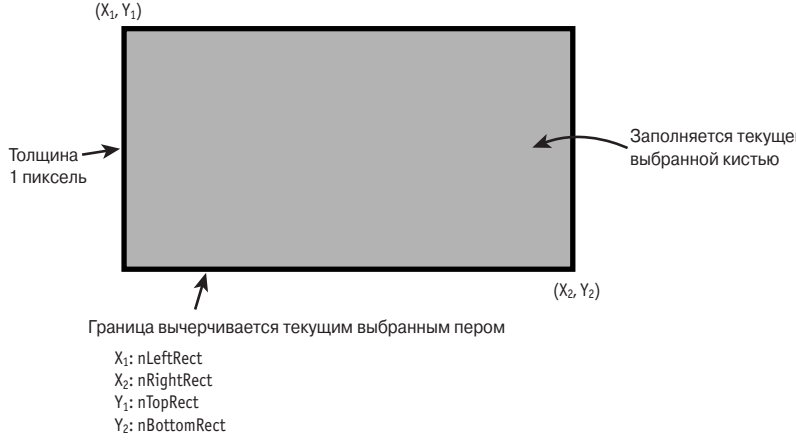


Рис. 4.5. Использование функции Rectangle()

НА ЗАМЕТКУ

Я хотел бы обратить ваше внимание на важную деталь. Координаты, передаваемые функции Rectangle(), представляют собой координаты границ прямоугольника. Это означает, что если стиль пера NULL и вы рисуете сплошной прямоугольник, то его размер с каждой стороны окажется на 1 пиксель меньше.

Кроме того, есть две дополнительные специальные функции для рисования прямоугольника — FillRect() и FrameRect().

```
int FillRect(HDC       hdc,   // Контекст устройства
        const RECT *lprc,     // Координаты прямоугольника
        HBRUSH     hbr);      // Дескриптор кисти

int FrameRect(HDC      hdc, // Контекст устройства
         const RECT *lprc,   // Координаты прямоугольника
         HBRUSH     hbr);    // Дескриптор кисти
```

Функция FillRect() рисует заполненный прямоугольник без граничных линий, включая верхний левый угол, но не включая нижний правый. Таким образом, если вы хотите заполнить прямоугольник с координатами (10,10) — (20,20), то должны передать функции структуру RECT с координатами (10,10) — (21,21). FrameRect() же рисует пустой прямоугольник с границей. Однако пути Microsoft неисповедимы, и вместо пера этой функции также передается кисть...

Вот пример вывода сплошного заполненного прямоугольника при помощи функции Rectangle():

```
// Создание пера и кисти
HPEN   blue_pen = CreatePen(PS_SOLID,1,RGB(0,0,255));
HBRUSH red_brush = CreateSolidBrush(RGB(255,0,0));

// Выбор пера и кисти
SelectObject(blue_pen);
SelectObject(red_brush);

// Вывод прямоугольника
Rectangle(hdc, 10, 10, 20, 20);

// Освобождение объектов
```

Вот аналогичный пример с использованием функции FillRect():

```
// Определяем прямоугольник
RECT rect {10, 10, 20, 20};

// Выводим прямоугольник
FillRect(hdc, &rect, CreateSolidBrush(RGB(255,0,0,));
```

Обратите внимание на то, что я определил и прямоугольник и кисть "на лету". Данную кисть не требуется удалять, поскольку она не выбирается в контекст графического устройства.

ВНИМАНИЕ

Во всех этих примерах я ничего не говорю об HDC и других деталях, полагая, что достаточно рассказал об этом раньше и вы в состоянии самостоятельно додумать опущенное. Очевидно, что для работы приведенных фрагментов вы должны иметь окно, контекст графического устройства и добавить соответствующий пролог и эпилог для корректной работы фрагмента. Далее в книге предполагается, что вы уже достаточно хорошо знакомы с ранее пройденным материалом, и больше таких предупреждений не будет.

На прилагаемом компакт-диске имеется демонстрационная программа DEMO4_3.CPP, в которой осуществляется вывод прямоугольников случайного размера и случайного цвета в произвольных позициях (рис. 4.6).

Окружность и эллипс

В начале 1980-х годов, чтобы нарисовать окружность, вам бы пришлось проявить смекалку и работать с ее формулой — $(x-x_0)^2+(y-y_0)^2=r^2$, либо воспользоваться параметрическим представлением окружности

$$\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases},$$

либо вместо вычислений осуществлять поиск в таблице. Теперь понятно, почему вывод окружности происходит медленнее вывода других графических объектов? Конечно, наличие процессоров Pentium III существенно упрощает задачу, но ее можно упростить и

иначе — воспользовавшись соответствующей функцией GDI. Правда, эта функция решает более общую задачу и выводит эллипс.

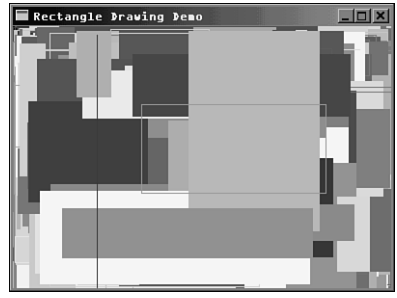


Рис. 4.6. Демонстрационная программа DEMO4_3.EXE

Если вы помните уроки геометрии, то должны знать, что окружность представляет собой частный случай эллипса с равными полуосями (ну или эллипс можно рассматривать как вытянутую вдоль одной оси окружность (или, как говаривал один преподаватель на военной кафедре, эллипс — это окружность, вписанная в квадрат с разными сторонами. :) — *Прим. ред.*)) Формула эллипса

$$\left(\frac{x-x_0}{a}\right)^2+\left(\frac{y-y_0}{b}\right)^2=1$$

вырождается в формулу окружности при $a=b=r$ (рис. 4.7).

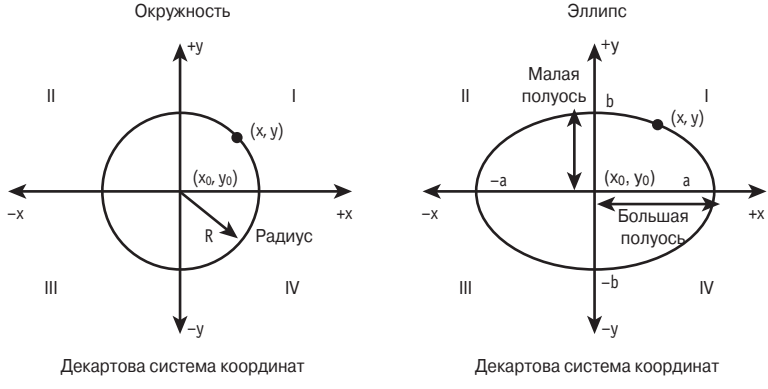


Рис. 4.7. Окружность и эллипс

Можно ожидать, что GDI будет использовать в качестве параметров эллипса величины его полуосей, однако это неверно: GDI для определения эллипса использует описанный вокруг него прямоугольник.

```
BOOL Ellipse(HDC hdc,     // Контекст устройства
    int nLeftRect,        // Левая граница прямоугольника
    int nTopRect,         // Верхняя граница прямоугольника
    int nRightRect,       // Правая граница прямоугольника
    int nBottomRect);     // Нижняя граница прямоугольника
```

Таким образом, для изображения приведенного на рис. 4.7 эллипса нужно вызвать функцию со следующими параметрами: Ellipse(hdc,-a,-b,a,b). Для того же, чтобы нарисо-

вать окружность с центром в точке (20, 20) и радиусом 10, следует осуществить вызов Ellipse(hdc,10,10,30,30).

На прилагаемом компакт-диске находится демонстрационная программа DEMO4_4.CPP, в которой функция Ellipse() используется для создания простой анимации. Этот тип анимации, осуществляемый как цикл стирания, перемещения и вывода изображения, очень похож на технологию двойной буферизации или переключения страниц, с которой мы будем иметь дело позже. Пока что вы можете повозиться с исходным текстом программы, пытаясь изменить его (попробуйте, например, добавить еще один эллипс).

Многоугольник

Последний примитив, с которым я познакомлю вас, — это многоугольник. Функция вывода Polygon() предназначена для быстрого вывода замкнутых многоугольников. Вот ее прототип:

```
BOOL Polygon(HDC      hdc,    // Контекст устройства
      CONST POINT *lpPoints, // Указатель на вершины
                             // многоугольника
      int      nCount);      // Количество вершин
```

Вы передаете функции список вершин многоугольника вместе с их количеством, и функция строит замкнутый многоугольник с использованием текущих пера и кисти (рис. 4.8).

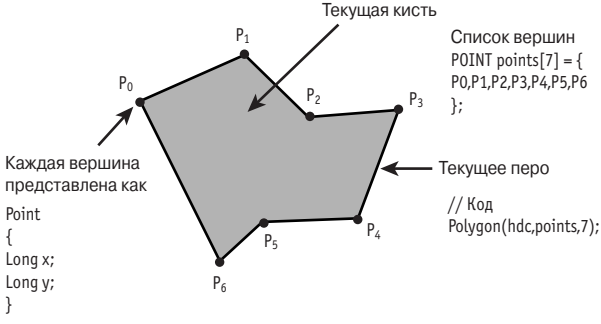


Рис. 4.8. Использование функции Polygon()

Вот пример использования функции Polygon():

```
// Создание многоугольника
POINT poly[7] = { p0x, p0y, p1x, p1y, p2x, p2y, p3x, p3y,
          p4x, p4y, p5x, p5y, p6x, p6y };

// Считая контекст устройства корректным, а кисть и перо -
// выбранными, выводим многоугольник
Polygon(hdc,poly,7);
```

На прилагаемом компакт-диске находится демонстрационная программа DEMO4_5.CPP, в которой осуществляется вывод случайных многоугольников с количеством вершин от 3 до 10 (рис. 4.9). Обратите внимание, что в силу случайности вершин многоугольники почти всегда вырождены (обладают самопересечениями). Можете ли вы предложить алгоритм генерации многоугольников, гарантирующий отсутствие самопересечений?

(В тексте программы есть ошибка, постарайтесь найти ее и исправить. Чтобы увидеть ее проявление, закомментируйте строку с вызовом Sleep(500), скомпилируйте программу,

запустите ее и немного подождите. Если вы не сможете найти ошибку, перечитайте раздел “Работа с перьями”. — *Прим. ред.*)

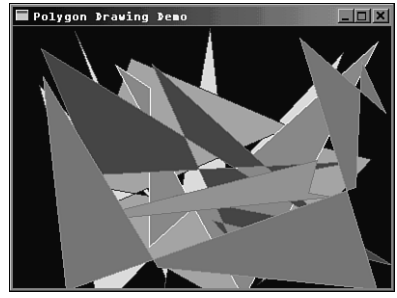


Рис. 4.9. Вывод многоугольников программой DEMO4_5.EXE

Текст и шрифты

Работа со шрифтами — исключительно сложная тема, которую я не намерен излагать в полном объеме. Если вас интересует эта тема, лучше всего обратиться к книге Чарльза Петцольда (Charles Petzold) *Programming Windows 95/98*. При разработке игр DirectX в большинстве случаев вывод текста осуществляется самостоятельно, при помощи собственных средств работы со шрифтами. Использование GDI для вывода текста обычно ограничивается выводом отладочной информации в процессе разработки. В конечном итоге для ускорения работы вашей игры вы будете вынуждены разработать собственную систему работы со шрифтами.

Тем не менее я постараюсь хотя бы показать, как изменить шрифт, с помощью которого выводят текст функции `DrawText()` и `TextOut()`. Это осуществляется путем выбора нового объекта шрифта в текущий контекст графического устройства, подобно тому как выбирается новое перо или кисть. В табл. 4.1 приведены некоторые константы, например `SYSTEM_FIXED_FONT` (которая представляет *моноширинный* шрифт, т.е. шрифт, ширина всех символов у которого одинакова. Символы *пропорциональных* шрифтов имеют разную ширину). Таким образом, для выбора нового шрифта в контекст графического устройства можно воспользоваться следующим вызовом:

```
SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

После этого вывод текста при помощи функций `DrawText()` и `TextOut()` будет осуществляться с помощью нового шрифта. Если же вам требуется бо́льшая свобода выбора шрифтов, можете выбирать один из встроенных шрифтов TrueType, перечисленных в табл. 4.4.

Таблица 4.4. Встроенные шрифты TrueType

Название шрифта	*Образец*
Courier New	`Hello, World!`
Courier New Bold	**`Hello, World!`**
Courier New Italic	*`Hello, World!`*
Courier New Bold Italic	***`Hello, World!`***
Times New Roman	Hello, World!

Название шрифта	*Образец*
Times New Roman Bold	**Hello, World!**
Times New Roman Italic	*Hello, World!*
Times New Roman Bold Italic	***Hello, World!***
Arial	Hello, World!
Arial Bold	**Hello, World!**
Arial Italic	*Hello, World!*
Arial Bold Italic	***Hello, World!***
Symbol	Ηελλο, Ωορλδ!

Для создания одного из этих шрифтов используется функция CreateFont().

```
HFONT CreateFont(
  int    nHeight,                // Логическая высота шрифта
  int    nWidth,                 // Логическая ширина шрифта
  int    nEscapement,            // Угол смещения
  int    nOrientation,           // Угол ориентации
  int    fnWeight,               // Вес шрифта
  DWORD  fdwItalic,              // Атрибут курсива
  DWORD  fdwUnderline,           // Атрибут подчеркивания
  DWORD  fdwStrikeOut,           // Атрибут перечеркивания
  DWORD  fdwCharSet,             // Набор символов
  DWORD  fdwOutputPrecision,     // Точность вывода
  DWORD  fdwClipPrecision,       // Точность обрезки
  DWORD  fdwQuality,             // Качество вывода
  DWORD  fdwPitchAndFamily,      // Наклон и семейство
  LPCTSTR lpszFace);             // Имя шрифта (табл. 4.4)
```

Объяснять все параметры этой функции слишком долго, так что я направлю вас за детальной информацией к справочной системе Win32 SDK и добавлю только то, что, создав шрифт, вы выбираете его дескриптор в текущий контекст графического устройства, после чего выводите текст выбранным шрифтом.

Таймер

Теперь рассмотрим работу со временем. Хотя эта тема может показаться не слишком важной, в видеоиграх она приобретает решающее значение. Без точной работы со временем и корректных задержек игра может оказаться слишком быстрой или чересчур медленной, с полной потерей иллюзии анимации.

Если вы помните, в главе 1, “Путешествие в пропасть”, упоминалось, что большинство игр имеют частоту кадров 30 fps, но до сих пор не рассматривалось, как можно этого добиться. В этом разделе вы узнаете о некоторых технологиях отслеживания времени и даже для отправления сообщений, базирующихся на значении времени. Позже в книге вы познакомитесь с тем, как рассмотренные идеи можно применить для того, чтобы обеспечить постоянную частоту кадров, и как быть в случае слишком медленных систем. Но первое, с чем мы познакомимся, — это сообщение WM_TIMER.

Сообщение WM_TIMER

В PC имеется встроенный таймер, который может работать с очень высокой точностью (в микросекундном диапазоне), но поскольку мы работаем в Windows, то пытаться самостоятельно работать с таймером — идея не из лучших. Вместо этого воспользуемся функциями, которые Windows предоставляет для работы со временем и таймером. Огромным преимуществом Windows является то, что она предоставляет возможность работы практически с неограниченным количеством виртуальных таймеров. Таким образом, программист может запускать и получать сообщения от ряда таймеров, несмотря на то что в системе всего лишь один физический таймер.

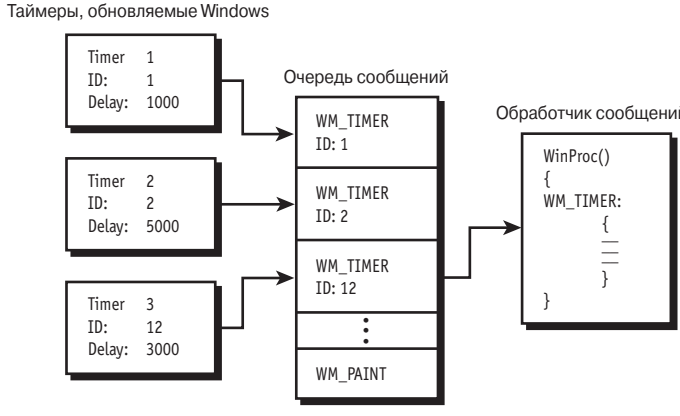


Рис. 4.10. Работа сообщений WM_TIMER

Когда вы создаете таймер, вы указываете иго идентификатор и время задержки, и таймер начинает посылать сообщения вашей процедуре окна с заданным интервалом. На рис. 4.10 показана одновременная работа нескольких таймеров, каждый из которых посылает сообщение WM_TIMER по истечении указанного интервала времени. Отличить сообщения одного таймера от сообщений другого можно по идентификатору таймера, указываемому при его создании. Вот функция, создающая таймер:

```
UINT SetTimer(HWND hwnd,  // Дескриптор родительского окна
 UINT     nIDevent,       // Идентификатор таймера
 UINT     nElapse,        // Временная задержка (миллисекунды)
 TIMERPROC lpTimerFunc);  // Функция обратного вызова таймера
```

Итак, для создания таймера вам требуется:

- дескриптор окна;
- идентификатор таймера, выбираемый вами;
- временной интервал (в миллисекундах).

Что касается последнего параметра функции, то он требует более детального пояснения. lpTimeFunc() представляет собой функцию обратного вызова наподобие WinProc; следовательно, вы можете создать таймер, который вместо отправления сообщений WM_TIMER будет вызывать указанную ему функцию обратного вызова. Лично я предпочитаю работать с сообщениями WM_TIMER и использую значение параметра lpTimeFunc, равное NULL.

Вы можете создать столько таймеров, сколько вам требуется, но не забывайте, что при этом расходуются ресурсы операционной системы. Если функция не в состоянии создать

таймер, она вернет значение 0; в противном случае возвращаемое значение — идентификатор созданного таймера.

Следующий вопрос состоит в том, как отличить сообщения одного таймера от сообщений другого. Для этого используется параметр сообщения wparam, который содержит идентификатор пославшего сообщение таймера. Вот пример создания двух таймеров, одного с периодом в 1 секунду и другого с периодом в 3 секунды:

```
#define TIMER_ID_1SEC  1
#define TIMER_ID_3SEC  2

// Следующий код может быть выполнен, например,
// при обработке сообщения WM_CREATE
SetTimer(hwnd, TIMER_ID_1SEC, 1000, NULL);
SetTimer(hwnd, TIMER_ID_3SEC, 3000, NULL);
```

Обратите внимание на то, что период таймера указывается в миллисекундах. Код обработчика сообщения таймера выглядит примерно так:

```
case WM_TIMER:
{
  // Какой из таймеров послал сообщение?
  switch(wparam)
  {
    case TIMER_ID_1SEC:
    {
    // Обработка сообщения таймера с периодом 1 секунда
    } break;

    case TIMER_ID_3SEC:
    {
    // Обработка сообщения таймера с периодом 3 секунды
    } break;

    default: break;
  } // switch

  return(0);
} break;
```

И наконец, по окончании работы с таймером мы должны удалить его при помощи функции

```
BOOL KillTimer(HWND hwnd,      // Дескриптор окна
        UINT uIDevent);        // Идентификатор таймера
```

Продолжая приведенный пример, вы можете удалить таймеры, скажем, при обработке сообщения WM_DESTROY.

```
case WM_DESTROY:
{
  // Удаляем таймеры
  KillTimer(TIMER_ID_1SEC);
  KillTimer(TIMER_ID_3SEC);

  PostQuitMessage(0);

} break;
```

ВНИМАНИЕ

Несмотря на всю кажущуюся доступность таймеров, не следует забывать, что они занимают ресурсы компьютера, в том числе и процессорное время. Поэтому не следует злоупотреблять таймерами; их надо удалять, как только они становятся больше не нужны.

На прилагаемом компакт-диске имеется демонстрационная программа DEMO4_6.CPP, в которой создаются три таймера с различными периодами и осуществляется вывод в окно при каждом сообщении от таймера.

И последнее: несмотря на то что период задается в миллисекундах, не следует ожидать от таймера точности, большей, чем 10–20 миллисекунд. Если вам требуется более высокая точность, воспользуйтесь высокопроизводительными таймерами Win32 или аппаратным счетчиком реального времени процессоров Pentium (с использованием ассемблерной инструкции RDTSC).

Низкоуровневая работа со временем

Создание таймеров представляет собой метод работы со временем, обладающий двумя недостатками: во-первых, таймеры отправляют сообщения, а во-вторых, таймеры не очень точны. В большинстве видеоигр требуется обеспечить вывод строго предопределенного количества кадров в секунду, а таймеры не так уж хорошо подходят для этой цели. Что нам действительно нужно — это способ точного получения системного времени. Win32 API предлагает для этой цели функцию

```
DWORD GetTickCount(void);
```

Эта функция возвращает количество миллисекунд после запуска Windows. Само по себе это значение не так уж полезно, но разность получаемых с помощью этой функции значений позволяет точно измерить интервал между событиями. Необходимо лишь вызвать ее в начале блока, запомнить полученное значение, и в конце цикла выполнить задержку на нужное время. Допустим, мы хотим обеспечить вывод ровно 30 кадров в секунду, т.е. один кадр должен выводиться 1/30 s = 33.33 ms. Следовательно, наш код вывода одного кадра должен выглядеть следующим образом:

```
// Получение начального момента времени
DWORD start_time = GetTickCount();

// Выполнение вычислений, вывод кадра и т.п.

// Ожидание, пока не пройдет 33 ms
while((GetTickCount() - start_time) < 33);
```

Вот о чем я твердил вам столько времени! Несмотря на то что в последнем цикле, по сути, зря расходуется процессорное время, такой метод обеспечивает строгую привязку выполнения участка кода ко времени.

НА ЗАМЕТКУ

Очевидно, что, если ваш компьютер не может уложиться в частоту 30 кадров в секунду, цикл будет занимать большее количество времени, чем 33 ms. Однако если компьютер может выполнять код со скоростью, например, от 30 до 100 кадров в секунду, то такая привязка обеспечит скорость вывода строго 30 кадров в секунду.

В качестве примера рассмотрите демонстрационную программу DEMO4_7.CPP, скорость вывода изображений в которой фиксирована и равна 30 кадрам в секунду. Вот обеспечивающий эту фиксацию код из функции WinMain():

```
// Получаем и сохраняем контекст графического устройства
hdc = GetDC(hwnd);
```

```
// Инициализируем генератор случайных чисел
srand(GetTickCount());

// Концы линии
int x1 = rand()%WINDOW_WIDTH;
int y1 = rand()%WINDOW_HEIGHT;
int x2 = rand()%WINDOW_WIDTH;
int y2 = rand()%WINDOW_HEIGHT;

// Начальная скорость каждого конца
int x1v = -4 + rand()%8;
int y1v = -4 + rand()%8;
int x2v = -4 + rand()%8;
int y2v = -4 + rand()%8;

// Главный цикл с использованием PeekMessage()
while(TRUE)
  {
  // Начальное время выполнения цикла
  DWORD start_time = GetTickCount();

  // Получение и обработка сообщений
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    if (msg.message == WM_QUIT)
       break;
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    } // if

    // Изменение цвета линии
    if (++color_change_count >= 100)
      {
      // Сброс счетчика
      color_change_count = 0;

      // Создание пера случайного цвета
      if (pen)
        DeleteObject(pen);
      pen = CreatePen(PS_SOLID,1,RGB(rand()%256,
                rand()%256,rand()%256));

      // Выбор пера в контекст графического устройства
      SelectObject(hdc,pen);

      } // if


    // Перемещение концов линии
    x1+=x1v;
    y1+=y1v;

    x2+=x2v;
    y2+=y2v;
```

```
    // Проверка соударения
    if (x1 < 0 || x1 >= WINDOW_WIDTH)
      {
      // Инвертирование скорости
      x1v=-x1v;
      x1+=x1v;
      } // if

    if (y1 < 0 || y1 >= WINDOW_HEIGHT)
      {
      // Инвертирование скорости
      y1v=-y1v;
      y1+=y1v;
      } // if

    // Проверка второго конца линии
    if (x2 < 0 || x2 >= WINDOW_WIDTH)
      {
      // Инвертирование скорости
      x2v=-x2v;
      x2+=x2v;
      } // if

    if (y2 < 0 || y2 >= WINDOW_HEIGHT)
      {
      // Инвертирование скорости
      y2v=-y2v;
      y2+=y2v;
      } // if

    // Вывод линии
    MoveToEx(hdc, x1,y1, NULL);
    LineTo(hdc,x2,y2);

    // Фиксация скорости вывода в 30 fps
    while((GetTickCount() - start_time) < 33);

    // Проверка выхода из программы
    if (KEYDOWN(VK_ESCAPE))
      SendMessage(hwnd, WM_CLOSE, 0,0);

  } // while


// Освобождение контекста графического устройства
ReleaseDC(hwnd,hdc);

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

} // WinMain
```

Кроме работы со временем, в данном коде имеется еще один важный момент, с которым нам еще придется встретиться: обработка столкновений. В демонстрационной программе мы перемещаем два конца линии, каждый из которых имеет свои координаты и скорость. В процессе перемещения концов отрезка в коде осуществляется проверка, не произошло ли столкновение концов отрезка с границами клиентской области. Если это так, то моделируется отражение конца отрезка от границы.

СЕКРЕТ

Если ваша задача — просто уменьшить скорость вывода графики кодом, можно воспользоваться функцией Sleep(), которой передается значение интервала задержки в миллисекундах. Например, чтобы задержать выполнение программы на 1 секунду, следует использовать вызов Sleep(1000).

Управляющие элементы

Дочерние управляющие элементы Windows в действительности представляют собой маленькие окна. Вот краткий список некоторых наиболее популярных управляющих элементов:

- статический текст;
- поля редактирования текста;
- кнопки;
- списки;
- полосы прокрутки.

Кроме того, имеется ряд типов кнопок:

- прямоугольные кнопки (push button);
- независимые переключатели (check box);
- круглые кнопки-переключатели (radio button).

Более того, у этих типов имеются свои подтипы. Однако большинство сложных управляющих элементов окон представляют собой набор базовых управляющих элементов. Например, управляющий элемент “каталог файлов” — это несколько списков, полей редактирования текста и кнопок. Если вы научитесь работать с перечисленными здесь типами управляющих элементов, то для вас не составит особого труда работа с управляющими элементами любой степени сложности. Более того, все управляющие элементы по сути одинаковы, и если вы научитесь работать с одним типом, то сможете работать и со всеми другими, в крайнем случае с подсказками из справочной системы Win32 SDK. В качестве такого одного типа управляющих элементов я выбрал кнопки.

Кнопки

Windows поддерживает большое количество самых разных типов кнопок. Думаю, все читатели этой книги знакомы с Windows, а потому детально рассказывать о том, что собой представляют кнопки, переключатели и другие управляющие элементы, не имеет никакого смысла. Так что сразу перейдем к вопросу о том, как создавать кнопки разных типов и как реагировать на сообщения от них. Для начала взгляните на табл. 4.5, в которой перечислены все доступные в Windows типы кнопок.

Для создания кнопки вы создаете окно с использованием "button" в качестве имени класса и с одним из стилей, перечисленных в табл. 4.5. Затем при работе с кнопкой она посылает сообщения WM_COMMAND вашему окну, как показано на рис. 4.11. Какая именно кнопка послала сообщение и что именно означает это сообщение, определяется параметрами wparam и lparam.

Таблица 4.5. Стили кнопок

Значение	*Описание*
BS_PUSHBUTTON	Прямоугольная кнопка, которая при выборе пользователем посылает окну-владельцу сообщение WM_COMMAND
BS_RADIOBUTTON	Маленькая круглая кнопка с текстом. По умолчанию текст выводится справа от кнопки
BS_CHECKBOX	Маленький переключатель с текстом. По умолчанию текст выводится справа от переключателя
BS_3STATE	Кнопка, внешне такая же, как и предыдущая, но может находиться в трех состояниях: выбрана, не выбрана и серая (не выбираемая)
BS_AUTOSTATE	Кнопка, внешне такая же, как и предыдущая, но при выборе пользователем циклически меняющая состояния — выбранная, серая, не выбранная
BS_AUTOCHECKBOX	Переключатель, который автоматически изменяет свое состояние на противоположное при выборе его пользователем
BS_AUTORADIOBUTTON	Круглая кнопка, отличающаяся тем, что при выборе Windows автоматически снимает состояние "выбранности" у всех других круглых кнопок той же группы
BS_OWNERDRAW	Кнопка, прорисовываемая владельцем. Окно-владелец получает сообщение WM_MEASUREITEM при создании кнопки и WM_DRAWITEM при изменении внешнего вида кнопки

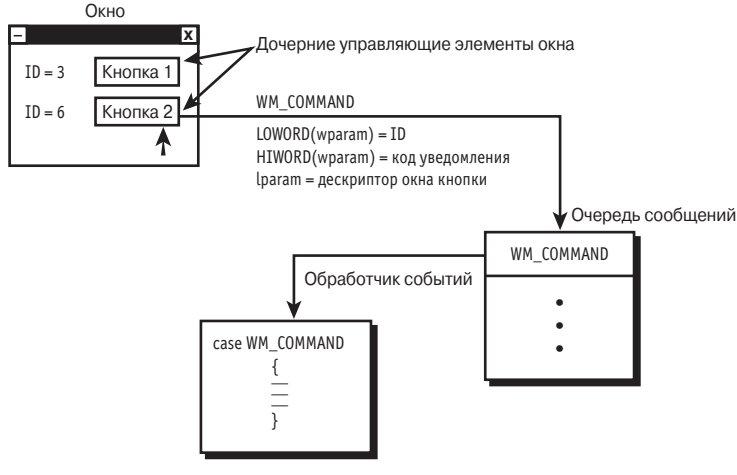


Рис. 4.11. Передача сообщения дочерним окном

Рассмотрим подробнее параметры, которые следует передать функции CreateWindowEx() для создания окна. Как уже отмечалось, имя класса должно быть "button". Флаг стиля должен иметь значение WS_CHILD|WS_VISIBLE, объединенный побитовым оператором ИЛИ с одним из стилей из табл. 4.5. Вместо дескриптора меню вы передаете выбранный вами идентификатор кнопки (конечно, с приведением типа к HMENU). В качестве примера приведу код создания прямоугольной кнопки с идентификатором, равным 100, и текстом "Push Me" на ней:

```
CreateWindowEx(NULL,
        "button",
```

```
"Push Me",
WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
10, 10, 100, 24,
main_window_handle,
(HMENU)100,
hinstance,
NULL);
```

Просто, не правда ли? Когда вы нажимаете кнопку, она посылает процедуре родительского окна сообщение WM_COMMAND со следующими параметрами:

LOWORD(wparam)	идентификатор дочернего окна
HIWORD(wparam)	код уведомления
lparam	дескриптор дочернего окна

Думаю, все понятно, кроме кода уведомления. Код уведомления описывает, что именно случилось с кнопкой, и принимает одно из значений, приведенных в табл. 4.6.

Таблица 4.6. Коды уведомлений кнопок

Код	*Значение*
BN_CLICKED	0
BN_PAINT	1
BN_HLITE	2
BN_UNHILITE	3
BN_DISABLE	4
BN_DOUBLECLICKED	5

Наиболее важными из перечисленных кодов уведомлений являются BN_CLICKED и BN_DOUBLECLICKED. Для работы с простой прямоугольной кнопкой можно использовать примерно такой код обработчика сообщения WM_COMMAND:

```
// Полагаем, что кнопка имеет идентификатор 100
case WM_COMMAND:
{
  // Проверка идентификатора
  if (LOWORD(wparam) == 100)
  {
    // Обрабатываем сообщение данной кнопки
  } // if

  // Обработка сообщений других дочерних
  // управляющих элементов, меню и т.п.

  return(0);
} break;
```

В качестве примера рассмотрите демонстрационное приложение DEMO4_8.CPP, в котором создается ряд различных кнопок и выводятся все сообщения от этих кнопок вместе со значениями wparam и lparam. На рис. 4.12 данная программа показана в действии. Поэкспериментируйте с ней, и вы получите более точное представление о том, как работают различные кнопки.

Рис. 4.12. Демонстрационное приложение DEMO4_8.EXE

Поработав с исходным текстом приложения DEMO4_8.CPP, вы очень скоро обнаружите, что хотя процедура окна позволяет обрабатывать сообщения от кнопок и вы можете реагировать на действия пользователя, вы не знаете, каким образом можно программно изменить состояние кнопок или как-то иначе воздействовать на них (кстати, ряд кнопок внешне выглядят просто как не реагирующие на действия с ними). Поскольку этот вопрос очень важен, рассмотрим его.

Отправка сообщений дочерним управляющим элементам

Поскольку дочерние управляющие элементы являются окнами, они могут получать сообщения так же, как и любые другие окна. Хотя они сами отправляют родительскому окну сообщения WM_COMMAND, им также можно посылать сообщения, которые будут обрабатываться соответствующей процедурой окна. Именно так и изменяются состояния любого управляющего элемента — путем отправки сообщений для него.

Имеется ряд сообщений, которые можно послать кнопкам с использованием функции SendMessage() и которые могут изменять состояния кнопок и/или получать информацию об их состоянии (вспомните о возвращаемом значении функции SendMessage()). Далее приводится список наиболее интересных сообщений и их параметров, посылаемых кнопкам.

Назначение: Эмуляция щелчка на кнопке
Сообщение: BM_CLICK
wparam 0
lparam 0

Пример:

```
// Эмуляция нажатия кнопки
SendMessage(hwndbutton, BM_CLICK, 0, 0);
```

Назначение: Изменение состояния переключателя или круглой кнопки
Сообщение: BM_SETCHECK
wparam fCheck
lparam 0

Значения fCheck могут быть следующими:

Значение	*Описание*
BST_CHECKED	Устанавливает выбранное состояние кнопки
BST_INDETERMINATE	Устанавливает неопределенное ("серое") состояние кнопки. Это значение используется только для стилей BS_3STATE и BS_AUTO3STATE
BST_UNCHECKED	Устанавливает невыбранное состояние кнопки

Пример:

```
// Устанавливаем выбранное состояние переключателя
SendMessage(hwndbutton, BM_SETCHECK, BST_CHECKED, 0);
```

Назначение: Получение состояния переключателя. Возможные возвращаемые значения показаны ниже

Сообщение: BM_GETCHECK
wparam 0
lparam 0

Значение	*Описание*
BST_CHECKED	Переключатель выбран
BST_INDETERMINATE	Неопределенное ("серое") состояние. Это значение возможно только для стилей BS_3STATE и BS_AUTO3STATE
BST_UNCHECKED	Переключатель не выбран

Пример:

```
// Получаем состояние переключателя
if(SendMessage(hwndbutton,BM_GETCHECK,0,0) == BST_CHECKED)
{
  // Переключатель выбран
} // if
else
{
  // Переключатель не выбран
} // else
```

Назначение: Используется для выделения кнопки, как если бы она была выбрана пользователем

Сообщение: BM_SETSTATE
wparam fState
lparam 0

Значение fState равно TRUE для выделения кнопки и FALSE — для его отмены.

Пример:

```
// Выделение кнопки
SendMessage(hwndbutton, BM_SETSTATE, 1, 0);
```

Назначение: Получение общего состояния кнопки. Возможные возвращаемые значения показаны ниже

Сообщение: BM_GETSTATE
wparam 0
lparam 0

Значение	*Описание*
BST_CHECKED	Кнопка выбрана
BST_FOCUS	Состояние фокуса ввода. Ненулевое возвращаемое значение указывает, что кнопка имеет фокус ввода с клавиатуры
BST_INDETERMINATE	Указывает, что переключатель находится в неопределенном состоянии. Это значение возможно только для стилей BS_3STATE и BS_AUTO3STATE
BST_PUSHED	Состояние выделенности кнопки. Ненулевое значение указывает, что кнопка выделена. Кнопка автоматически выделяется при позиционировании курсора над ней и нажатии левой кнопки мыши. Выделение снимается при освобождении левой кнопки мыши
BST_UNCHECKED	Указывает, что кнопка не выбрана

Пример:

```
// Этот код можно использовать для получения состояния кнопки
switch(SendMessage(hwndbutton, BM_GETSTATE, 0, 0))
{
  //Действия при разных состояниях
  case BST_CHECKED:      {} break;
  case BST_FOCUS:        {} break;
  case BST_INDETERMINATE: {} break;
  case BST_PUSHED:       {} break;
  case BST_UNCHECKED:    {} break;
  default:               break;
} // switch
```

На этом завершим рассмотрение дочерних управляющих элементов. Надеюсь, основная идея понятна, а с деталями можно познакомиться в справочной системеWin32SDK.

Получение информации

При разработке игры жизненно важное значение приобретает информация о системе, в которой эта игра работает. Как и следовало ожидать, Windows предоставляет ряд функций для получения массы информации об установках Windows и аппаратном обеспечении, на котором работает операционная система.

В Win32 для этой цели имеется ряд функций Get*(), а в DirectX — функций GetCaps*(). Я ограничусь только некоторыми функциями Win32, которые буду время от времени использовать в данной книге. С функциями DirectX, которые в большей степени связаны с функционированием мультимедиа, вы встретитесь в следующей части книги.

В этом разделе вы познакомитесь с тремя основными функциями, хотя на самом деле их во много раз больше. Пожалуй, нет ничего, что бы вы хотели узнать о Windows и не могли бы этого сделать при помощи той или иной функции Get*(), но подробнее о них вы можете узнать из справочной системы Win32.

Первая функция, которую я хочу представить на ваше рассмотрение, — GetSystemInfo(). С ее помощью можно получить практически любую информацию о работе аппаратного обеспечения — типе процессора, их количестве и т.д. Вот ее прототип:

```
VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);
```

Функция получает в качестве параметра указатель на структуру SYSTEM_INFO и заполняет ее поля соответствующей информацией. Вот как выглядит эта структура:

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    };
  };
  DWORD  dwPageSize;
  LPVOID lpMinimumApplicationAddress;
  LPVOID lpMaximumApplicationAddress;
  DWORD  dwActiveProcessorMask;
  DWORD  dwNumberOfProcessors;
  DWORD  dwProcessorType;
  DWORD  dwAllocationGranularity;
  WORD   wProcessorLevel;
  WORD   wProcessorRevision;
} SYSTEM_INFO;
```

К сожалению, описание всех полей данной структуры выходит за рамки нашей книги, так что я вынужден прибегнуть к любимой рекомендации — обратиться к справочной системе Win32 SDK. Здесь я упомяну только, что поле dwNumberOfProcessors указывает количество процессоров в системе, а dwProcessorType — тип процессора, который может принимать следующие значения на платформе PC:

```
#define PROCESSOR_INTEL_386     386
#define PROCESSOR_INTEL_486     486
#define PROCESSOR_INTEL_PENTIUM  586
```

Следующая функция, с которой я хочу вас познакомить, позволяет получить массу информации о Windows и десктопе:

```
int GetSystemMetrics(int nIndex);
```

Это очень мощная функция. Вы передаете ей в качестве параметра индекс данных, которые хотели бы получить, и она возвращает вам искомое. Для этой функции я сделаю исключение и опишу в табл. 4.7 некоторые из индексов данных, которые могут быть переданы функции.

Таблица 4.7. Возможные параметры функции GetSystemMetrics()

Значение	*Описание*
SM_CLEANBOOT	Каким образом выполнялась загрузка операционной системы: 0 — нормальная загрузка 1 — безопасный режим 2 — безопасный режим с сетью
SM_CMOUSEBUTTONS	Количество кнопок у мыши (0, если мыши в системе нет)
SM_CXBORDER SM_CYBORDER	Ширина и высота границы окна в пикселях
SM_CXCURSOR SM_CYCURSOR	Ширина и высота курсора в пикселях

Значение	*Описание*
SM_CXDOUBLECLICK SM_CYDOUBLECLICK	Ширина и высота в пикселях области, внутри которой должны находиться места двух последовательных щелчков мышью, чтобы быть воспринятым как один двойной щелчок (кроме того, два щелчка должны выполняться в пределах отрезка времени предопределенной протяженности)
SM_CXFULLSCREEN SM_CYFULLSCREEN	Ширина и высота клиентской области окна, распахнутого на весь экран
SM_CXICON SM_CYICON	Ширина и высота пиктограммы, принятая по умолчанию. Обычно 32×32 пикселя
SM_CXMAXIMIZED SM_CYMAXIMIZED	Размеры в пикселях максимизированного окна верхнего уровня
SM_CXMENUSIZE SM_CYMENUSIZE	Размеры в пикселях кнопок полосы меню
SM_CXMIN SM_CYMIN	Минимальные размеры окна в пикселях
SM_CXMINIMIZED SM_CYMINIMIZED	Размеры в пикселях нормального минимизированного окна
SM_CXSCREEN SM_CYSCREEN	Размеры экрана в пикселях
SM_MOUSEPRESENT	Ненулевое значение при наличии в системе мыши; если мышь отсутствует, возвращает значение 0
SM_MOUSEWHEELPRESENT	Только для Windows NT: ненулевое значение при наличии в системе мыши с колесиком
SM_NETWORK	При наличии сети младший бит установлен; остальные биты зарезервированы для использования в будущем
SM_SWAPBUTTON	Ненулевое значение, если в установках Windows левая и правая кнопки мыши обменены местами

В таблице перечислены не все возможные индексы, так что совет почаще обращаться к справочной системе Win32 SDK остается в силе и для функции GetSystemMetrics(). Вот пример использования данной функции для создания окна размером на весь экран:

```
if (!(hwnd = CreateWindowEx(NULL,
                WINDOW_CLASS_NAME,
                "Button demo",
                WS_POPUP|WS_VISIBLE,
                0, 0,
               GetSystemMetrics(SM_CXSCREEN),
               GetSystemMetrics(SM_CYSCREEN),
                NULL, NULL, hinstance, NULL)))
  return (0);
```

НА ЗАМЕТКУ Обратите внимание на использование флага WS_POPUP вместо флага WS_OVERLAPPEDWINDOW. Таким образом создается окно без рамки и управляющих элементов, и в результате мы получаем полностью пустой экран, что, собственно, и требуется для полноэкранной игры.

В качестве другого примера в программе может быть использован код, который проверяет наличие мыши в системе:

```
if (GetSystemMetrics(SM_MOUSEPRESENT))
{
  // Мышь имеется
}
else
{
  // Мыши нет
}
```

И наконец, когда мы выводим текст, нам может понадобиться информация о текущем шрифте, например чтобы знать ширину и высоту символов и общую длину выводимой строки и корректно позиционировать ее на экране. Для получения этой информации можно воспользоваться функцией

```
BOOL GetTextMetrics(HDC hdc,
        LPTEXTMETRIC lptm);
```

Не удивляйтесь тому, что данной функции требуется передавать контекст графического устройства; вспомните, что у нас может быть много контекстов графических устройств, причем у каждого из них может быть свой выбранный шрифт. Параметр lptm представляет собой указатель на заполняемую функцией структуруTEXTMETRIC:

```
typedef struct tagTEXTMETRIC
{
 LONG tmHeight;            // Высота шрифта
 LONG tmAscent;            // Надстрочный элемент
 LONG tmDescent;           // Подстрочный элемент
 LONG tmInternalLeading;   // Внутренний междустрочный интервал
 LONG tmExternalLeading;   // Внешний междустрочный интервал
 LONG tmAveCharWidth;      // Средняя ширина
 LONG tmMaxCharWidth;      // Максимальная ширина
 LONG tmWeight;            // Вес шрифта
 LONG tmOverhang;          // Выступ
 LONG tmDigitizedAspectX;  // Сжатие по оси x
 LONG tmDigitizedAspectY;  // Сжатие по оси y
 BYTE tmFirstChar;         // Первый символ шрифта
 BYTE tmLastChar;          // Последний символ шрифта
 BYTE tmDefaultChar;       // Символ, использующийся вместо отсутствующих
 BYTE tmBreakChar;         // Символ разрыва
 BYTE tmItalic;            // Шрифт курсивный
 BYTE tmUnderlined;        // Шрифт подчеркнутый
 BYTE tmStruckOut;         // Шрифт перечеркнутый
 BYTE tmPitchAndFamily;    // Семейство шрифта
 BYTE tmCharSet;           // Набор символов шрифта
} TEXTMETRIC;
```

Большинство из нас не работают в печати и не имеют ежедневно дела со шрифтами, так что многие термины могут оказаться просто непонятными. Взгляните на рис. 4.13: возможно, он прояснит некоторые неясности.

Вот пример того, как описанная функция используется для центрирования текста:

```
TEXTMETRIC tm;

GetTextMetrics(hdc, &tm);
```

```
// Используем данные tm для центрирования строки по
// горизонтали исходя из ширины окна WINDOW_WIDTH
int x_pos = WINDOW_WIDTH -
   strken("Center this string")*tm.tmAveCharWidth/2;

// Вывод текста в соответствующей позиции
TextOut(hdc,x_pos,0, "Center this string",
      strlen("Center this string"));
```

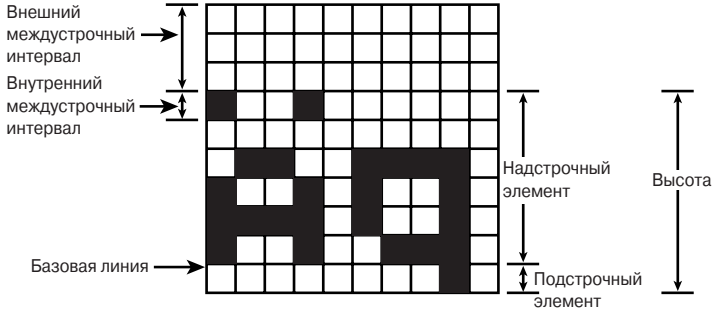


Рис. 4.13. Составные части символов

T3D Game Console

В начале книги упоминалось о том, что программирование игр Windows/DirectX очень похоже на 32-битовое программирование для DOS, если создать оболочку приложения Windows и структуру кода, которая скрывает детали работы с Windows. К этому моменту вы узнали достаточно, чтобы воплотить это на практике, поэтому сейчас можно приступить к созданию заготовки игры T3D Game Console, которая послужит основой всех демонстрационных программ и игр в оставшейся части книги.

Вы уже знаете, что для создания приложения Windows необходимо создать процедуру окна и функцию WinMain(). Мы создадим минимальное приложение Windows, имеющее эти компоненты и создающее обобщенное окно. Затем это приложение вызывает три функции, которые и реализуют логику игры. В результате детали обработки сообщений Windows и другие вопросы работы Win32 остаются за кадром и вам не придется иметь с ними дело (если только по каким-либо причинам вы не захотите этого сами). На рис. 4.14 показана архитектура T3D Game Console.

Как видите, для реализации игры нам требуются только три функции:

```
int Game_Init    (void* parms = NULL, int num_parms = 0);
int Game_Main    (void* parms = NULL, int num_parms = 0);
int Game_Shutdown(void* parms = NULL, int num_parms = 0);
```

- Game_Init() вызывается (причем только один раз) перед входом в главный цикл событий в WinMain(). Здесь выполняется вся инициализация вашей игры.
- Game_Main() напоминает функцию main() в обычной программе C/C++, за исключением того, что она вызывается каждый раз после проверки наличия (и при необходимости обработки) сообщений Windows. Именно эта функция и представляет собой логику игры. Физическая модель игры, искусственный интеллект и т.п. — все это реализовано либо в функции Game_Main(), либо в функциях, которые она

вызывает. Единственное предостережение: эта функция должна выводить очередной кадр и завершаться для того, чтобы обработчик сообщений мог выполнять свою работу. В связи с этим необходимо обратить внимание на временность автоматических переменных. Если какие-то данные должны сохраняться между кадрами, их следует хранить в глобальных переменных (либо локальных static-переменных функции Game_Main()).

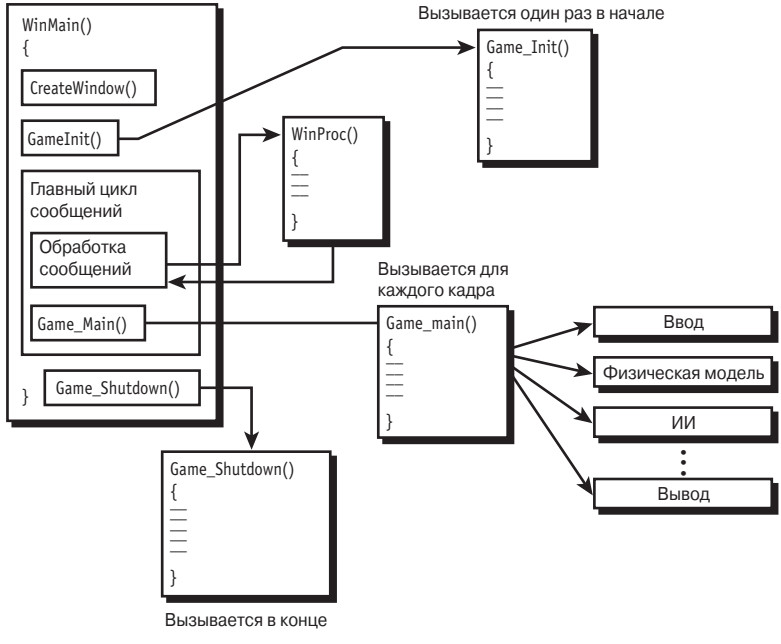


Рис. 4.14. Архитектура T3D Game Console

- Game_Shutdown() вызывается по окончании работы главного цикла WinMain() (при получении сообщения WM_QUIT). В этой функции выполняются все действия по освобождению ресурсов и завершению игры.

Исходный текст игры T3D Game Console находится на прилагаемом компакт-диске в файле T3DCONSOLE.CPP. Ниже приведен код функции WinMain(), в которой и осуществляются вызовы функций игры.

```
// WINMAIN ////////////////////////////////////////////
int WINAPI WinMain( HINSTANCE hinstance,
                    HINSTANCE hprevinstance,
                    LPSTR lpcmdline,
                    int ncmdshow)
{

WNDCLASSEX winclass;  // Класс окна
HWND       hwnd;      // Дескриптор окна
MSG        msg;       // Сообщения
HDC        hdc;       // Контекст графического устройства

// Определение класса
```

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

// Сохранение hinstance
hinstance_app = hinstance;

// Регистрация класса
if (!RegisterClassEx(&winclass))
   return(0);

// Создание окна
if (!(hwnd = CreateWindowEx(NULL,
                    WINDOW_CLASS_NAME,
                    "T3D Game Console Version 1.0",
                    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                    0,0,400,300,
                    NULL, NULL,
                    hinstance,
                    NULL)))
   return(0);

// Сохранение дескриптора окна
main_window_handle = hwnd;

// Инициализация игры
Game_Init();

// Главный цикл событий
while(TRUE)
   {
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
      {
      if (msg.message == WM_QUIT)
         break;

      TranslateMessage(&msg);

      DispatchMessage(&msg);
      } // if

      // Вывод очередного кадра игры
      Game_Main();
```

```
  } // while

// Завершение игры
Game_Shutdown();

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

} // WinMain
```

Пересмотрите приведенный код. Мы уже не раз встречались с подобным кодом в наших демонстрационных программах. Разница только в наличии вызовов функций Game_*(), заготовки которых представлены ниже.

```
//////////////////////////////////////////////////////////

int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
  SendMessage(main_window_handle,WM_CLOSE,0,0);

// Успешное завершение (или ваш собственный код ошибки)
return(1);

} // Game_Main

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

int Game_Init(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз после создания окна
// и входа в главный цикл обработки сообщений. Здесь
// выполняются все инициализирующие действия.

// Успешное завершение (или ваш собственный код ошибки)
return(1);

} // Game_Init

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

int Game_Shutdown(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз при завершении работы
// игры. В ней выполняются все действия по освобождению
// захваченных ресурсов и т.п.

// Успешное завершение (или ваш собственный код ошибки)
return(1);

} // Game_Shutdown
```

Да, сейчас эти функции выполняют не слишком-то много работы. И это правильно — кодом вы будете заполнять их в каждой вашей игре. Все, что я позволил себе, — это обеспечить выход из игры по нажатию пользователем клавиши <Esc> для завершения работы игры.

Думаю, вы обратили внимание на список параметров каждой из функций Game_*():

```
int Game_*(void *parms = NULL, int num_parms = 0);
```

Параметр num_parms введен для удобства, чтобы при необходимости вместе с параметрами вы могли передать и их количество, а тип первого параметра void* выбран из соображений гибкости. Однако никто не заставляет вас использовать именно эти параметры функций (и вообще их использовать). При желании вы можете изменить их.

И наконец, вы можете решить, что мне следовало бы создавать окно без управляющих элементов и размером на весь экран, воспользовавшись стилем WS_POPUP. Я бы мог сделать это, но мне кажется, что для демонстрационных программ лучше иметь оконную версию, поскольку ее проще отлаживать. Кроме того, изменения, которые необходимо внести для получения полноэкранной версии, не существенны и вы можете сделать это в любой момент самостоятельно.

C++

Если вы программист на C, то синтаксис объявления int Game_Main(void *parms = NULL, int num_parms = 0); может несколько озадачить вас. В C++ такое присвоение в объявлении называется значением параметра по умолчанию. Эти значения подставляются вместо отсутствующих в реальном вызове параметров. Например, в нашем случае вызовы Game_Main() или Game_Main(NULL) эквивалентны вызову Game_Main(NULL,0) (при этом, разумеется, вы можете передавать и корректные параметры, например вызывая Game_Main(&list,12)). Более подробную информацию и правила работы с параметрами по умолчанию вы найдете в любом учебнике по C++.

Если вы запустите приложение T3DCONSOLE.EXE, то не увидите ничего, кроме пустого окна. Все, что нужно для его заполнения, — написать код функций Game_*() и получить за это много зелененьких бумажек. :-)

Чтобы продемонстрировать, как это делается (добавляется код, а не получаются бумажки), я написал простенькую программку DEMO4_9.CPP. Для GDI получилась совсем неплохая программка — окно с летящими звездами. Посмотрите на ее код и попробуйте увеличить или уменьшить скорость ее работы. В программе также продемонстрирована концепция цикла анимации, состоящего из операций стирания, перемещения и вывода изображения. Частота кадров фиксирована и составляет 30 fps.

Резюме

Теперь, мой юный джедай, ты можешь считать себя властелином окон — по крайней мере в степени, достаточной для борьбы с империей программирования игр. В этой главе вы познакомились с огромным количеством материала: GDI, управляющие элементы, работа со временем, получение информации. Все эти знания позволят приступить к созданию серьезных приложений Windows.

В следующей части вы познакомитесь с волнующим и захватывающим миром DirectX.

ЧАСТЬ II

DirectX и основы двухмерной графики

Глава 5

Основы DirectX и COM 195

Глава 6

Первое знакомство с DirectDraw 219

Глава 7

Постигаем секреты DirectDraw и растровой графики 257

Глава 8

Растеризация векторов и двухмерные преобразования 365

Глава 9

DirectInput 499

Глава 10

DirectSound и DirectMusic 541

ГЛАВА 5

Основы DirectX и COM

В этой главе рассматривается DirectX и компоненты, составляющие эту потрясающую технологию. Кроме того, здесь детально описывается COM (Component Object Model — модель составных объектов), на базе которой созданы все компоненты DirectX. Если вы обычный программист на языке C, то вам необходимо уделить этой главе особое внимание. Но не волнуйтесь, я постараюсь изложить материал в доступной форме.

Однако сразу предупреждаю: пока не дочитаете эту главу до конца, не делайте никаких выводов о том, поняли вы ее или нет. DirectX и COM тесно взаимосвязаны, поэтому сложно объяснить работу одной из технологий без знания другой. Попробуйте представить, как бы вы объяснили понятие нуля, не используя в его определении слова нуль. Если вы думаете, что это просто, то вы ошибаетесь!

Вот основные вопросы, которые будут затронуты в этой главе.

- Введение в DirectX
- Модель составных объектов (COM)
- Примеры реализации COM
- Насколько DirectX и COM стыкуются друг с другом
- Будущее COM

Азы DirectX

Я начинаю чувствовать себя проповедником Microsoft (намек в сторону Microsoft: шлите мои 30 сребреников), пытаясь переманить всех читателей на их сторону. Но у плохих парней всегда лучшие технологии, не правда ли? Что бы вы предпочли для участия в драке: один из истребителей, предназначенных для уничтожения имперских суперзвезд, или какой-нибудь наполовину переделанный транспорт повстанцев? Понятно, о чем я говорю?

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

На рис. 5.1 показаны схемы разработки игры для Windows с использованием DirectX и без нее. Обратите внимание, насколько ясным и элегантным решением является DirectX.

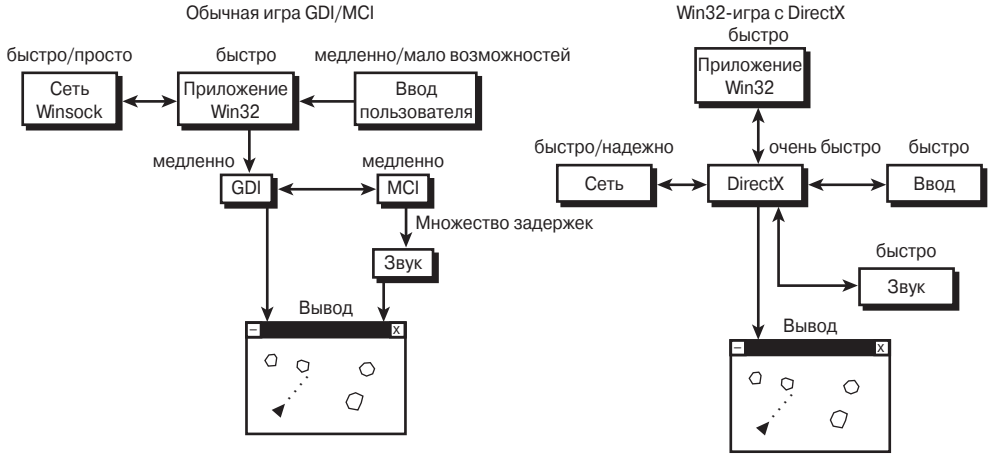


Рис. 5.1. DirectX против GDI/MCI

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

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

В настоящий момент DirectX состоит из ряда компонентов. Их список приводится ниже и показан на рис. 5.2.

- DirectDraw (нет в DirectX 8.0+)
- DirectSound
- DirectSound3D
- DirectMusic
- DirectInput
- DirectPlay
- DirectSetup

- Direct3DRM
- Direct3DIM
- DirectX Graphics (объединение DirectDraw и Direct3D)
- DirectX Audio (объединение DirectSound и DirectMusic)
- DirectShow

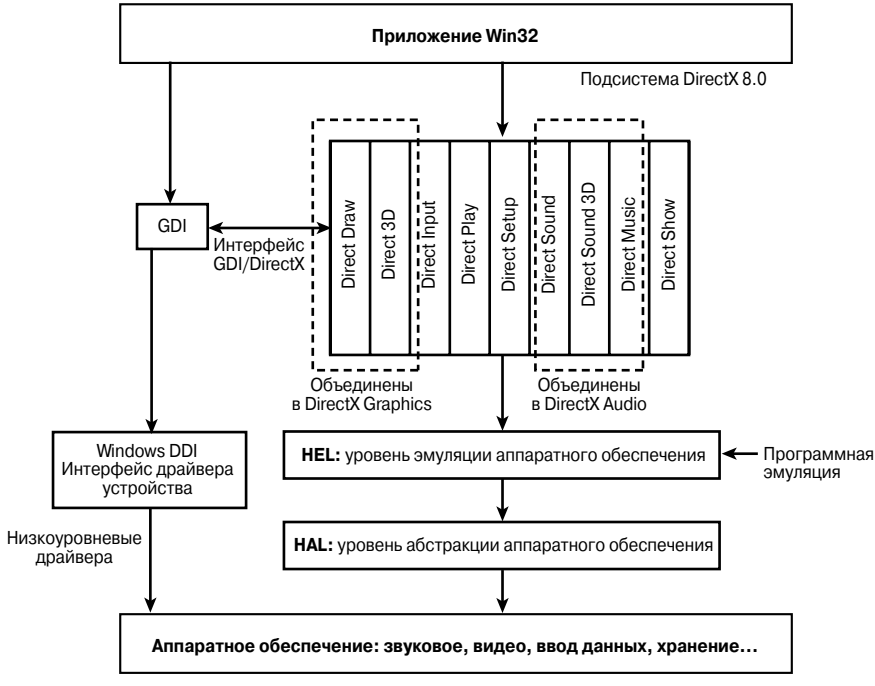


Рис. 5.2. Архитектура DirectX и отношения с Win32

В версии DirectX 8.0 компания Microsoft решила тесно интегрировать DirectDraw и Direct3D, назвав это объединение DirectX Graphics. В результате этого решения DirectDraw удален из DirectX 8.0, хотя возможность использовать DirectDraw осталась. Просто этот компонент в DirectX 8.0 остался необновленным. Кроме того, теперь тесно интегрированы DirectSound и DirectMusic и то, что получилось, называется DirectX Audio. Наконец, в DirectX интегрирован компонент DirectShow (который ранее представлял собой часть DirectMedia). Эти парни из Microsoft действительно завалены работой доверху!

Все это может показаться несколько лишним и сбивающим с толку, но самое приятное в DirectX то, что благодаря возможностям COM при желании можно использовать DirectX 3.0, DirectX 5.0, DirectX 6.0 или любую другую версию, которая нам нравится и которая отвечает нашим потребностям. В этой книге версии 7.0 или какой-нибудь из 8.0 будет вполне достаточно. Кроме того, если вы знакомы с какой-либо из версий DirectX, значит, вы знакомы и со всеми остальными. Если синтаксис и изменился, то ненамного. Интерфейс изменился чуть больше, но, вообще говоря, все версии DirectX примерно одинаковы. Единственное, что постоянно меняется, — это Direct3D, но здесь мы не будем обсуждать эту тему, ведь нас интересуют не “внутренности” DirectX, а программирование игр. Вы изучите DirectX достаточно хорошо для того, чтобы создавать игры, но программирование игр — это не только программирование DirectX.

HEL и HAL

На рис. 5.2 можно заметить, что DirectX основывается на двух уровнях, которые называются HEL (Hardware Emulation Layer — уровень эмуляции аппаратного обеспечения) и HAL (Hardware Abstraction Layer — уровень абстракции аппаратного обеспечения). Так как DirectX создан с дальним прицелом в будущее, предполагается, что в этом будущем аппаратное обеспечение будет поддерживать дополнительные возможностями, которыми можно будет пользоваться при работе с DirectX. Ну, а что делать, пока аппаратное обеспечение не поддерживает эти возможности? Проблемы этого рода и призваны решать HEL и HAL.

HAL находится ближе к "железу" и общается с устройствами напрямую. Обычно это драйвер устройства, написанный производителем, и с помощью обычных запросов DirectX вы обращаетесь непосредственно к нему. Отсюда вывод: HAL используется тогда, когда вам надо обратиться к функции, поддерживаемой самим устройством (что существенно ускоряет работу). Например, если вы хотите загрузить растровый рисунок на экран, то аппаратно это будет сделано гораздо быстрее, чем при использовании программного цикла.

HEL используется тогда, когда устройство не поддерживает необходимую вам функцию. Например, вы хотите заставить изображение на экране вращаться. Если видеоадаптер не поддерживает вращение на аппаратном уровне, на арену выходит HEL и выполнение этой функции берет на себя программное обеспечение. Понятно, что этот способ работает медленнее, но худо-бедно, а ваша программа продолжает работать. При этом вы даже не знаете, на каком именно уровне выполнено ваше задание. Если задача решается на уровне HAL, это будет сделано на аппаратном уровне. В противном случае для выполнения вашего задания будет вызвана программа из HEL.

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

Подробнее о базовых классах DirectX

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

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

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

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

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

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

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

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

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

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

DirectSetup/AutoPlay. Квазикомпоненты DirectX, обеспечивающие инсталляцию DirectX на машину пользователя непосредственно из вашего приложения и автоматический запуск игры при вставке компакт-диска в дисковод. DirectSetup представляет собой небольшой набор функций, которые загружают файлы DirectX на компьютер пользователя во время запуска вашей программы и заносят все необходимые записи в системный реестр. AutoPlay — это обычная подсистема для работы с компакт-дисками, которая ищет в

корневом каталоге компакт-диска файл Autoplay.inf. Если этот файл обнаружен, то AutoPlay запускает команды из этого файла.

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

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

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

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

COM: дело рук Microsoft… или врага рода человеческого?

Современные программы зачастую содержат миллионы строк программного кода, и очень скоро в больших системах счет пойдет уже на миллиарды. Для таких систем абстрагирование и иерархичность крайне важны — в противном случае не избежать хаоса.

Одним из последних веяний в информационных технологиях стало использование объектно-ориентированных языков программирования, где с этой точки зрения два наиболее "продвинутых" продукта — C++ и Java. Язык C++ является объектно-ориентированным потомком C. В то же время Java, хотя и основан на C++, представляет собой полностью объектно-ориентированный язык, в котором объектная ориентированность проявляется

более последовательно и строго. К тому же Java — это, скорее, платформа, в то время как C++ — просто язык программирования.

В любом случае это замечательные языки, и все зависит от того, насколько правильно вы их используете. Увы, несмотря на то что C++ буквально напичкан различными возможностями объектно-ориентированного программирования, многие программисты не используют их, а если и используют — то неправильно. Из-за этого разработка огромных программ все еще представляет большие проблемы. Но это далеко не все, для чего предназначена технология COM.

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

При работе с платами компьютера наиболее приятно то, что, добавляя новую плату, вы не должны отчитываться об этом перед другими платами. Как вы понимаете, в случае программного обеспечения ситуация несколько сложнее. Программу придется по крайней мере перекомпилировать. И это вторая проблема, решить которую призвана COM. Нам требуется возможность расширять возможности COM-объектов так, чтобы программы, работавшие со старой версией объекта, продолжали успешно работать и с новой версией. Кроме того, можно изменить COM-объекты, не перекомпилируя при этом саму программу, — а это очень большое преимущество.

Поскольку вы можете заменять старые COM-объекты новыми, не перекомпилируя при этом программу, вы можете обновлять ваше программное обеспечение на машине пользователя, не создавая никаких очередных "заплаток" или новых версий программ. Например, у вас есть программа, которая использует три COM-объекта: один отвечает за графику, один за звук и один за работу в сети (рис. 5.3). А теперь представьте, что вы уже продали 100 000 копий своей программы и хотите избежать отсылки 100 000 новых версий. При использовании COM для обновления работы с графикой вам достаточно дать своим пользователям новый COM-объект, и старая программа будет использовать его вместо старого; при этом не нужно делать решительно ничего: ни перекомпилировать, ни компоновать приложение. Конечно, реализация такой технологии на уровне программирования — задача очень сложная. Чтобы создать собственный COM-объект, требуется приложить массу усилий. Но зато как легко будет потом им пользоваться!

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

Что такое COM-объект

В действительности COM-объект представляет собой класс (или набор классов) на языке C++, который реализует ряд *интерфейсов* (которые, в свою очередь, являются наборами функций). Эти интерфейсы используются для связи с COM-объектами. Взгляните на рис. 5.4. На нем вы видите простой COM-объект с тремя интерфейсами: `IGRAPHICS`, `ISOUND` и `IINPUT`.

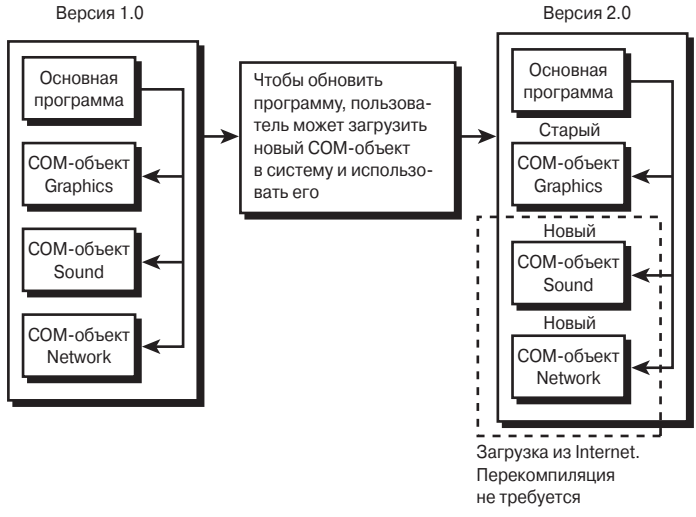


Рис. 5.3. Представление о COM

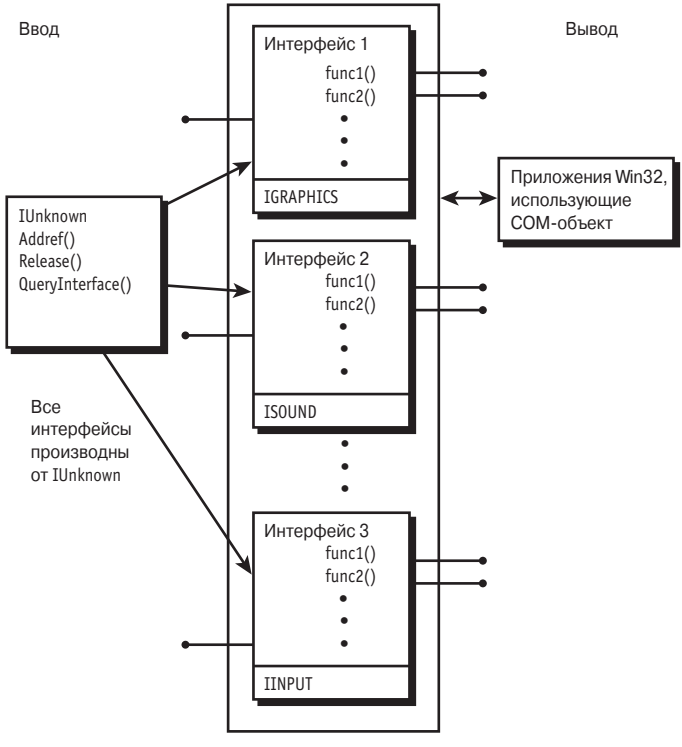


Рис. 5.4. Интерфейсы COM-объекта

У каждого из этих трех интерфейсов есть свой набор функций, который вы можете (если знаете, как) использовать в своей работе. Итак, каждый COM-объект может иметь один или несколько интерфейсов и у вас может быть один или несколько COM-объектов. В соответствии со спецификацией COM все интерфейсы, созданные вами,

должны быть производными от специального базового класса IUnknown. Для вас, как для программиста на C++, это означает, что IUnknown — это некая отправная точка, с которой нужно начинать создание интерфейсов.

Давайте взглянем на определение класса IUnknown:

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

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

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

НА ЗАМЕТКУ

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

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

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

Кроме того, одно из правил COM гласит: если у вас есть интерфейс, то вы можете получить доступ к любому другому интерфейсу, так как они все происходят из одного COM-объекта. В общих черта это означает: где бы вы ни находились, вы можете попасть куда захотите (рис. 5.5).

СОВЕТ

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

Довольно любопытна функция AddRef(). В COM-объектах используется технология, называемая счетчиком ссылок (reference counting) и отслеживающая все ссылки на объекты. Необходимость в этом объясняется одной из особенностей COM-технологии: она не ориентируется на конкретный язык программирования. Следовательно, когда создается COM-объект или интерфейс, для того чтобы отследить, сколько имеется ссылок на

этот объект, вызывается AddRef(). Если бы COM-объект использовал вызовы malloc() или new[] — это было бы явным требованием использовать для разработки конкретный язык программирования C или C++. Когда счетчик ссылок обнуляется, объект автоматически уничтожается.

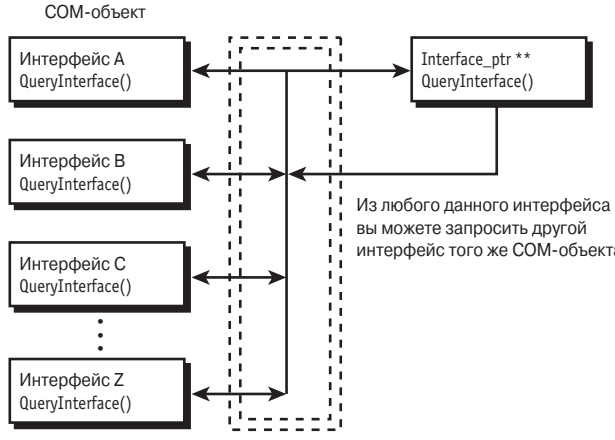


Рис. 5.5. Работа с интерфейсами COM-объекта

Тут же возникает второй вопрос: если COM-объекты — это классы, написанные на C++, то каким образом их можно создавать или использовать в Visual Basic, Java, ActiveX и т.д.? Просто так уж сложилось, что создатели для реализации COM использовали виртуальные классы C++. Вас никто не заставляет использовать именно этот язык программирования. Главное, чтобы созданный вами машинный код был аналогичен создаваемому компилятором Microsoft C++ при создании виртуальных классов. Тогда созданный COM-объект будет вполне корректен и работоспособен.

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

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

Еще об идентификаторах интерфейсов и GUID

Как я уже упоминал, каждый COM-объект и интерфейс должен обладать уникальным 128-битовым идентификатором, который используется для доступа к объекту. Такие идентификаторы носят название GUID (Globally Unique Identifiers — глобально уникальные идентификаторы). В более узком понимании, говоря о COM-интерфейсах, они называются идентификаторами интерфейсов (Interface ID — IID). Чтобы получить такие идентификаторы, вы можете воспользоваться программой GUIDEN.EXE, созданной Microsoft (или любой другой программой, созданной для этих целей). На рис. 5.6 показана работа этой программы.

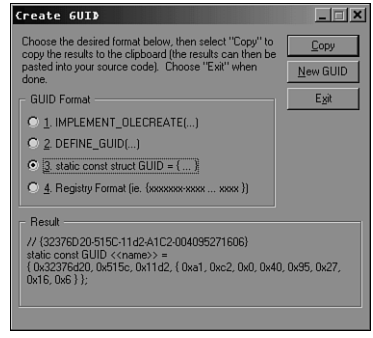


Рис. 5.6. Генератор GUID GUIDGEN.EXE в действии

Все, что вам необходимо сделать, — это выбрать, в каком виде вы хотите получить GUID (существует четыре различных формата). После этого программа генерирует 128-битовый вектор, который никогда не будет сгенерирован опять, нигде и никогда. Выглядит неправдоподобно? Это просто математика и теория вероятности, так что не забивайте себе голову такими мелочами. Главное, что это работает.

После того как вы сгенерировали GUID, скопируйте этот номер в буфер и вставьте его в программу. Вот пример IID, который я сгенерировал, когда писал этот параграф:

```
// {C1BCE961-3E98-11D2-A1C2-004095271606}
static const <<name>> =
{ 0xc1bce961, 0x3e98, 0x11d2,
{ 0xa1, 0xc2, 0x00, 0x40, 0x95, 0x27, 0x16, 0x06}};
```

В программе, конечно, следует заменить `<<name>>` реальным именем GUID.

Идентификаторы интерфейсов используются для указания на COM-объекты и их интерфейсы. Итак, когда бы и где бы не был создан новый COM-объект или набор интерфейсов, этот идентификатор — единственное, что нужно знать вашей программе для работы с объектом.

Создание COM-объекта

Создание полноценных COM-объектов в данной книге рассматриваться не будет. Вам необходимо знать лишь то, как с ними работать. Однако если вы хоть немного похожи на меня, то наверняка захотите узнать побольше. Итак, давайте все же создадим очень простой объект, который поможет ответить на некоторые возникающие вопросы.

Вы уже знаете, что все COM-объекты содержат интерфейсы. При этом все объекты должны порождаться из класса IUnknown. После того как вы создадите все интерфейсы, вы вкладываете их в классы-контейнеры. В качестве примера создадим COM-объект с тремя интерфейсами: ISound, IGraphics и IInput. Вот как можно их определить:

```
// Графический интерфейс
struct IGraphics : IUnknown
{
   virtual int InitGraphics (int mode) = 0;
   virtual int SetPixel (int x, int y, int c) = 0;
   // другие методы...
};

// Звуковой интерфейс
struct ISound : IUnknown
```

```
{
   virtual int InitSound (int driver) = 0;
   virtual int PlaySound (int note, int vol) = 0;
   // другие методы...
};

// Интерфейс ввода
struct IInput : IUnknown
{
   virtual int InitInput (int device) = 0;
   virtual int ReadStick (int stick) = 0;
   // другие методы...
};
```

Теперь, когда все интерфейсы описаны, создадим класс-контейнер, который является сердцем COM-объекта.

```
class CT3D_Engine: public IGraphics, ISound, IInput
{
public:

   // Реализация IUnknown
   virtual HRESULT __stdcall QueryInterface
         (const IID &iid, void **ip) {/* Реализация */}

   // Этот метод увеличивает счетчик ссылок на 1
   virtual ULONG __stdcall AddRef()    {/* Реализация */}

   // Этот метод уменьшает счетчик ссылок на 1
   virtual ULONG __stdcall Release()   {/* Реализация */}

   // Обратите внимание, что метод для создания
   // объекта отсутствует

   // Реализация каждого из интерфейсов

   // IGraphics
   virtual int InitGraphics (int mode) {/* Реализация */}

   virtual int SetPixel (int x, int y, int c)
                           {/* Реализация */}

   // ISound
   virtual int InitSound (int driver)  {/* Реализация */}

   virtual int PlaySound (int note, int vol)
                           {/* Реализация */}

   // IInput
   virtual int InitInput (int device)  {/* Реализация */}

   virtual int ReadStick (int stick)   {/* Реализация */}

private:
   //  Локальные закрытые переменные и функции
};
```

НА ЗАМЕТКУ

Обратите внимание: вопрос о создании COM-объекта так и остался не освещен. Это действительно целая проблема. В спецификации указано, что существует множество путей создания COM-объектов, но ни один из них не привязывает конкретную реализацию к той или иной платформе или языку. Один из самых простых путей — это создание функции `CoCreateInstance()` или `ComCreate()`, которые создают первоначальный экземпляр объекта класса `IUnknown`. Эти функции обычно загружают DLL, в которой содержится COM-класс, и после этого ваша программа уже способна работать с этим классом. Но, повторяю, эта технология лежит за рамками того, что вам нужно знать. Просто мне очень захотелось изложить все это здесь, чтобы вы были хоть немного в курсе дела. Но продолжим разбирать наш пример.

Как вы видите из примера, COM-интерфейсы, как и сама программа, — всего лишь виртуальные классы C++ с некоторыми особенностями. Однако настоящий COM-объект должен быть не только правильно создан, но и зарегистрирован в системном реестре. Кроме того, следует выполнить еще ряд правил. Однако на самом нижнем уровне это всего лишь классы со своими методами (или, если вы программист на C, структуры с указателями на функции).

А сейчас давайте сделаем маленькое отступление и еще раз посмотрим, что мы знаем о COM.

Краткое повторение COM

COM — это новый способ написания компонентов программного обеспечения, позволяющий создавать повторно используемые программные модули, которые динамически подключаются во время выполнения программы. Интерфейсы — это не более чем коллекция методов и функций, на которые ссылаются указатели из таблицы виртуальных функций (подробнее об этом будет сказано в следующем разделе).

Каждый COM-объект и интерфейс отличается от других своим глобальным идентификатором GUID, который вы должны генерировать для своих COM-объектов и интерфейсов. Вы используете GUID, чтобы обращаться к COM-объектам и интерфейсам и совместно использовать их с другими программистами.

Если вы создаете новый COM-объект, который должен будет заменить старый, то наряду с новыми интерфейсами, которые вы можете добавить в новый объект, следует поддерживать все старые интерфейсы. Это очень важное правило! Все программы, работающие с COM-объектом, должны работать с новой версией этого объекта без перекомпиляции.

COM — это всего лишь общая спецификация, которая может быть реализована на любом языке и на любом компьютере. Единственное правило: бинарный код COM-объекта должен быть таким же, как бинарный код виртуального класса, полученного с помощью Microsoft VC. При этом, если соблюдались правила создания COM-объектов, они могут быть использованы и на других вычислительных платформах, таких, как Mac, SGI и т.д.

COM открыла возможность построения сверхбольших программ (с миллиардами строк) путем использования компонентных архитектур. DirectX, OLE и ActiveX основаны на технологии COM. Именно поэтому вы должны понимать принципы ее работы.

Пример COM-программы

Примером создания полноценного COM-объекта и его интерфейсов может служить демонстрационная программа `DEMO5_1.CPP`. Программа реализует COM-объект `CCOM_OBJECT`, который состоит из двух интерфейсов: `IX` и `IY`. Эта программа — довольно приличная реализация COM-объекта. Конечно, в ней пропущены такие высокоуровневые моменты, как создание DLL, динамическая загрузка и т.д. Но этот объект полностью реализован, так как описаны все его методы и в качестве базового класса используется `IUnknown`.

Я хочу, чтобы вы взглянули на программу повнимательнее, поэкспериментировали с ее кодом и прочувствовали, как все это работает. В листинге 5.1 показан полный исходный текст COM-объекта, а также простая тестовая программа, в которой этот объект создается и используется.

Листинг 5.1. Работа с COM-объектом

```
// DEMO5_1.CPP — очень маленький пример COM-программы
// Примечание: не полностью соответствует спецификации COM

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

#include <stdio.h>
#include <malloc.h>
#include <iostream.h>
#include <objbase.h>

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

// Все GUID сгенерированы с помощью программы GUIDGEN.EXE

// {B9B8ACE1-CE14-11d0-AE58-444553540000}
const IID IID_IX =
   { 0xb9b8ace1, 0xce14, 0x11d0,
   { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 }};


// {B9B8ACE2-CE14-11d0-AE58-444553540000}
const IID IID_IY =
   { 0xb9b8ace2, 0xce14, 0x11d0,
   { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 }};

// {B9B8ACE3-CE14-11d0-AE58-444553540000}
const IID IID_IZ =
   { 0xb9b8ace3, 0xce14, 0x11d0,
   { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 }};

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

// Определение интерфейса IX
interface IX: IUnknown
{
   virtual void __stdcall fx(void)=0;
};

// Определение интерфейса IY
interface IY: IUnknown
{
   virtual void __stdcall fy(void)=0;
};

//////////////////////////////////////////////////////////
```

```
// Определение COM-объекта
class CCOM_OBJECT : public IX,
                    public IY
{
public:

   CCOM_OBJECT() : ref_count(0) {}
   ~CCOM_OBJECT() {}

private:

   virtual HRESULT __stdcall QueryInterface(
                const IID &iid, void **iface);
   virtual ULONG   __stdcall AddRef();
   virtual ULONG   __stdcall Release();

   virtual void    __stdcall fx(void)
      {cout << "Вызвана функция fx." << endl; }
   virtual void    __stdcall fy(void)
      {cout << "Вызвана функция fy." << endl; }

   int ref_count;

};

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

HRESULT __stdcall CCOM_OBJECT::QueryInterface(
                        const IID &iid, void **iface)
{
   // Функция приводит указатель this или указатель на
   // IUnknown к указателю на запрошенный интерфейс

   // Обратите внимание на сравнение с GUID
   // в начале функции

   // запрос интерфейса IUnknown
   if (iid==IID_IUnknown)
   {
      cout << "Запрошен интерфейс IUnknown" << endl;
      *iface = (IX*)this;
   } // if


   // Запрос IX?
   if (iid==IID_IX)
   {
      cout << "Запрошен интерфейс IX" << endl;
      *iface = (IX*)this;
   } // if
   else  // Запрос IY?
   if (iid==IID_IY)
   {
```

```
      cout << "Запрошен интерфейс IY" << endl;
      *iface = (IY*)this;
   } // if
   else
   { // Не найден!
      cout << "Запрошен неизвестный интерфейс!" << endl;
      *iface = NULL;
      return(E_NOINTERFACE);
   } // if

   // если запрашиваемый интерфейс найден, возвращаем
   // указатель на IUnknown и вызываем AddRef()
   ((IUnknown *)(*iface))->AddRef();

   return(S_OK);

} // QueryInterface

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

ULONG __stdcall CCOM_OBJECT::AddRef()
{
   // Увеличение счетчика ссылок
   cout << "Добавление ссылки" << endl;
   return(++ref_count);
} // AddRef

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

ULONG __stdcall CCOM_OBJECT::Release()
{
   // Уменьшение счетчика ссылок
   cout << "Удаление ссылки" << endl;
   if (--ref_count==0)
   {
      delete this;
      return(0);
   } // if
   else
      return(ref_count);
} // Release

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

IUnknown *CoCreateInstance(void)
{
   // Минимальная реализация CoCreateInstance().
   // Создает экземпляр COM-объекта;, здесь
   // я решил преобразовать указатель к IX.
   // Точно так же может быть использован IY.

   IUnknown *comm_obj = (IX *)new(CCOM_OBJECT);
   cout << "Создание объекта Comm" << endl;
```

```
   // Обновление счетчика ссылок
   comm_obj->AddRef();

   return(comm_obj);

} // CoCreateInstance

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

void main(void)
{

   // Создание COM-объекта
   IUnknown *punknown = CoCreateInstance();

   // создание двух NULL-указателей на интерфейсы IX и IY
   IX *pix=NULL;
   IY *piy=NULL;

   // Запрос указателя на интерфейс IX
   punknown->QueryInterface(IID_IX, (void **)&pix);

   // Попытка выполнения метода IX
   pix->fx();

   // Удаление интерфейса
   pix->Release();

   // Запрос указателя на интерфейс IY
   punknown->QueryInterface(IID_IY, (void **)&piy);

   // Попытка выполнения метода IX
   piy->fy();

   // Удаление интерфейса
   piy->Release();

   // Удаление самого COM-объекта
   punknown->Release();

} // main
```

Откомпилированная программа находится на прилагаемом компакт-диске (файл DEMO5_1.EXE). Если вы захотите самостоятельно поэкспериментировать с программой, не забывайте, что вы должны создавать консольное приложение Win32.

Работа с COM-объектами DirectX

Теперь, когда у вас есть общее представление о том, что такое DirectX и как работает COM, давайте подробнее рассмотрим, как они работают вместе. Как уже отмечалось, DirectX состоит из множества COM-объектов. При загрузке DirectX эти COM-объекты

находятся в вашей системе в виде DLL-файлов. Что же произойдет, если вы запускаете DirectX-игру стороннего разработчика, а одна или несколько DLL уже загружены другими DirectX-приложениями, которые используют некоторые интерфейсы и их методы?

При написании программ есть свои тонкости. Разработчики DirectX знали, что будут иметь дело с программистами игр, и понимали, что большинство из них ненавидят программировать в Windows. Они знали, что лучше максимально спрятать COM-начинку от программиста, иначе DirectX никогда не станет популярной технологией программирования. Таким образом, работа с COM-объектами в DirectX на 90% скрыта за вызовами функций. Вам не придется самостоятельно вызывать `CoCreateInstance()`, инициализировать COM и делать прочие неприятные вещи. Тем не менее, возможно, вам придется вызывать `QueryInterface()` для того, чтобы получить новый интерфейс, но к этому мы вернемся позже. Главное в том, что DirectX скрывает от вас работу с COM-объектами и вы можете сосредоточить свои усилия на использовании функциональных возможностей DirectX.

С учетом сказанного становится понятно, что для того, чтобы скомпилировать DirectX-программу, вы должны подключить ряд библиотек, представляющих собой надстройку над COM — инструменты и функции, с помощью которых DirectX создает COM-объекты и работает с ними. В большинстве случаев вам будет достаточно следующих библиотек:

```
DDRAW.LIB
DSOUND.LIB
DINPUT.LIB
DINPUT8.LIB
DSETUP.LIB
DPLAYX.LIB
D3DIM.LIB
D3DRM.LIB
```

Однако учтите — эти библиотеки не содержат COM-объектов. Это всего лишь инструментальные библиотеки, которые, в свою очередь, обращаются к COM-объектам. Когда вы вызываете COM-объект DirectX, в результате вы получаете указатель на какой-нибудь интерфейс, который и используете для выполнения действий. Например, в программе `DEMO5_1.CPP` после получения указателя на интерфейс вы можете вызывать его функции, или методы. Если вы программист на языке C и чувствуете дискомфорт при работе с указателями на функции, скорее читайте следующий раздел; если же вы программист на C++, то следующий раздел можете смело пропустить.

COM и указатели на функции

После создания COM-объекта вы получаете указатель на интерфейс, который на самом деле представляет собой указатель VTABLE (Virtual Function Table — указатель на таблицу виртуальных функций) (рис. 5.7). Основное свойство COM и виртуальных функций совпадает и заключается в том, что связывание и определение того, какая именно функция должна быть вызвана, происходит в процессе выполнения программы. В C++ эта возможность является встроенной, но и в C можно достичь того же эффекта, используя указатели на функции.

Указатель на функцию — это специальный тип указателя, который обеспечивает возможность вызова функций. Если разные функции имеют один и тот же прототип, то мы можем вызвать любую из них, используя указатель на такую функцию. Представим, например, что вы хотите создать функцию графического драйвера, которая должна выводить точку на экран. Предположим также, что ваш драйвер должен быть рассчитан на работу с различными видеокартами, как показано на рис. 5.8.

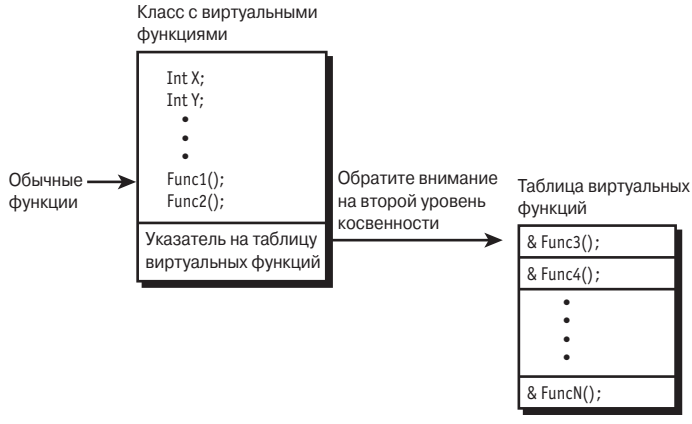


Рис. 5.7. Архитектура таблицы виртуальных функций

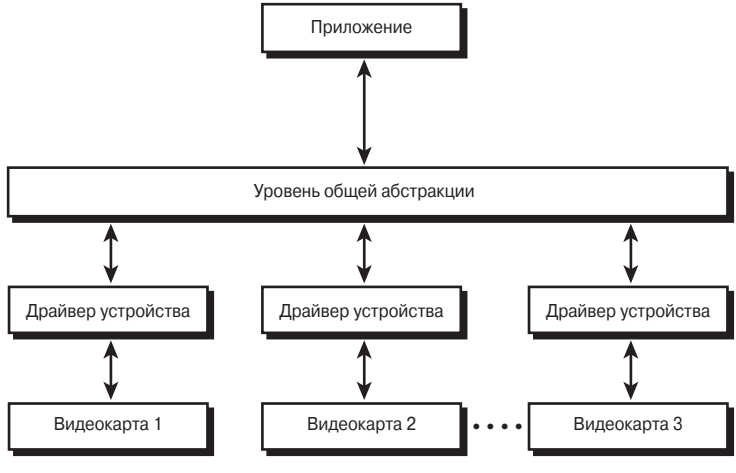


Рис. 5.8. Модель поддержки различных видеокарт

Вам требуется, чтобы функция работала одинаково для любой из видеокарт, но при этом необходимо учесть, что внутренний код для работы с каждой конкретной картой будет отличаться от кода, работающего с другой видеокартой. Вот типичный пример решения данной проблемы на C:

```
int SetPixel( int x, int y, int card)
{
  // Какая у нас видеоплата?
  switch (card)
    {
      case ATI:   {/* Код для ATI   */} break;
      case VOODOO: {/* Код для VOODOO */} break;
      case SIII:  {/* Код для SIII  */} break;
    .
    .
    .
      default:    {/* Код для VGA   */} break;
```

```
    } // switch
  // Завершение функции
  return(1);
} // SetPixel
```

Вы поняли, в чем проблема? Оператор switch не подходит для решения данной задачи. Он медленно работает, громоздкий и располагает к ошибкам, которые вы легко можете внести в функцию при добавлении поддержки новой видеокарты. Гораздо лучшим решением было бы использование указателей на функции:

```
// Объявление указателя на функцию
int ( *SetPixel)(int x, int y, int color);

// Функции для вывода точек разными видеокартами
int SetPixel_ATI(int x, int y, int color)
{
  // Тело функции для ATI
} // SetPixel_ATI

int SetPixel_VOODOO(int x, int y, int color)
{
  // Тело функции для VOODOO
} // SetPixel_VOODOO

int SetPixel_SIII(int x, int y, int color)
{
  // Тело функции для SIII
} // SetPixel_SIII
```

Теперь при запуске системы она проверяет, какая видеокарта установлена в компьютере, и затем — один и только один раз! — устанавливает указатель на функцию соответствующей видеокарты. Например, если вы хотите, чтобы SetPixel() указывала на функцию, работающую с платами ATI, в вашей программе должен быть следующий фрагмент кода (рис. 5.9):

```
// Определение указателя на функцию
SetPixel = SetPixel_ATI;
```

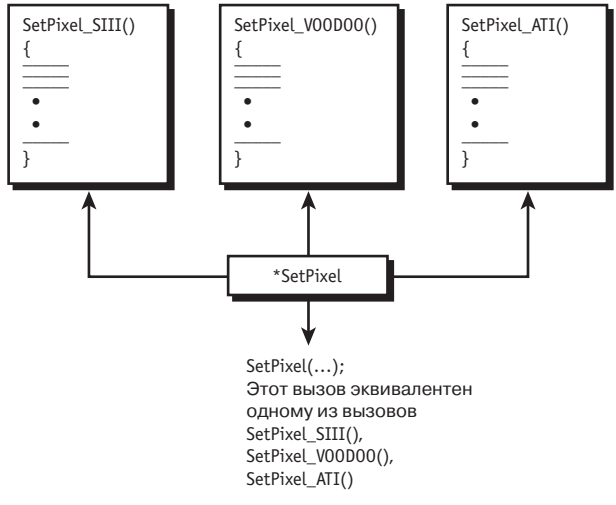


Рис. 5.9. Использование указателей на функции

Теперь, после этого присвоения, SetPixel() представляет собой не что иное, как псевдоним SetPixel_ATI(). Это ключевой момент для понимания концепции указателя на функцию. Теперь вы можете просто вызывать функцию SetPixel(), передавая ей те же параметры, что и функции SetPixel_ATI():

```
// Реально вызывается SetPixel_ATI(10,20,4);
SetPixel(10,20,4);
```

Итак, при том, что вы не изменяете вашу программу, в зависимости от конкретного значения указателя на функцию она работает по-разному. Это настолько удачная технология, что многое в C++ и в виртуальных функциях основано на ней. Собственно говоря, это и есть механизм виртуальных функций — позднее связывание указателя с функцией, встроенное в язык.

С учетом сказанного давайте закончим разрабатываемый драйвер. Итак, сначала нужно протестировать систему и выяснить, какая видеокарта в ней установлена, а затем присвоить указателю SetPixel соответствующую видеокарте функцию SetPixel*().

```
int SetCard(int Card)
{
  // Установка указателя на функцию в соответствии
  // с имеющейся видеокартой
  switch(card)
  {
    case ATI:
    {
      SetPixel=SetPixel_ATI;
    } break;
    case VOODOO:
    {
      SetPixel=SetPixel_VOODOO;
    } break;
    case SIII:
    {
      SetPixel=SetPixel_SIII;
    } break;
    default: break;
  } // switch
} // SetCard
```

Теперь в начале программы достаточно просто вызвать функцию настройки:

```
SetCard(card);
```

Надеюсь, теперь вы понимаете, как работают указатели на функции и виртуальные функции, и мы можем приступить к рассмотрению того, как эти технологии используются в DirectX.

Создание и использование интерфейсов DirectX

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

Прежде всего для того, чтобы экспериментировать с DirectDraw, нам нужно следующее.

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

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

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

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

Эта функция возвращает базовый интерфейс DirectDraw 1.0. В главе 6, "Первое знакомство с DirectDraw", параметры этой функции рассматриваются более подробно, а пока достаточно знать, что эта функция создает объект DirectDraw и возвращает указатель на интерфейс в переменной lpdd.

После этого можно обращаться к DirectDraw. Хотя нет — ведь вы же не знаете, какие методы и функции вам доступны, именно поэтому вы и читаете эту книгу! В качестве примера установим видеорежим 640×480 с 256 цветами.

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

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

По существу, любой вызов DirectX выполняется следующим образом:

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

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

Запрос интерфейсов

Как ни странно, но в DirectX номера версий не синхронизированы, и это создает определенные проблемы. Когда вышла первая версия DirectX, интерфейс DirectDraw назывался IDIRECTDRAW. Когда вышла вторая версия DirectX, DirectDraw был обновлен до версии 2.0 и мы получили интерфейсы IDIRECTDRAW и IDIRECTDRAW2. В версии 6.0 набор интерфейсов расширился до IDIRECTDRAW, IDIRECTDRAW2 и IDIRECTDRAW4. После выхода версии 7.0 к ним добавился интерфейс IDIRECTDRAW7, который остается последним, так как в версии DirectX 8.0 DirectDraw не поддерживается.

Вы спрашиваете, что случилось с третьей и пятой версией? Понятия не имею! Но в результате, даже если вы используете DirectX 8.0, это еще не означает, что все интерфейсы обновлены до этого же номера версии. Более того, разные интерфейсы могут иметь раз-

ные версии. Например, DirectX 6.0 может иметь интерфейс DirectDraw версии 4.0 — IDIRECTDRAW4, но DirectSound при этом имеет версию 1.0 и его интерфейс имеет имя IDIRECTSOUND. Кошмар!.. Мораль сей басни такова: когда вы используете интерфейс DirectX, вы должны убедиться в том, используется ли самая последняя его версия. Если вы в этом не уверены, воспользуйтесь указателем на интерфейс версии 1.0, получаемый при создании объекта, чтобы получить указатель на интерфейс последней версии.

Непонятно? Вот конкретный пример. DirectDrawCreate() возвращает указатель на базовый интерфейс первой версии, но нас интересует интерфейс DirectDraw под именем IDIRECTDRAW7. Как же нам получить все возможности интерфейса последней версии?

Существует два пути: использовать низкоуровневые функции COM или получить указатель на интересующий нас интерфейс при помощи вызова QueryInterface(), что мы сейчас и сделаем. Порядок действий следующий: сначала создается COM-интерфейс DirectDraw с помощью вызова DirectDrawCreate(). При этом мы получаем указатель на интерфейс IDIRECTDRAW. Затем, используя этот указатель, мы вызываем QueryInterface(), передавая ему идентификатор интересующего нас интерфейса, и получаем указатель на него. Вот как выглядит соответствующий код:

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

// В DDRAW.H находим идентификатор интерфейса IDIRECTDRAW7
// и используем его для получения указателя на интерфейс
lpdd->QueryInterface(IID_IDirectDraw7, &lpdd7);

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

// Присвоение значения NULL для безопасности
lpdd=NULL;

// Работа с интерфейсом IDIRECTDRAW7

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

Теперь вы знаете, как получить указатель на один интерфейс с помощью другого. В DirectX 7.0 Microsoft добавила новую функцию DirectDrawCreateEx(), которая сразу возвращает интерфейс IDIRECTDRAW7 (и после этого тут же, в DirectX 8.0, Microsoft полностью отказывается от DirectDraw...).

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

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

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

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

Будущее COM

Сегодня существует ряд объектных технологий, схожих с COM, например CORBA. Однако, поскольку вы работаете над играми в среде Windows, знать о других технологиях вам не положено :).

Последняя версия COM называется COM++. Это еще более надежная реализация COM, с улучшенными правилами и множеством тщательно продуманных деталей. COM++ позволяет еще легче создавать компонентное программное обеспечение. Конечно, COM++ несколько сложнее, чем COM, но такова жизнь!

В дополнение к COM и COM++ существует Internet/intranet-версия COM — DCOM (Distributed COM — распределенная COM). При использовании этой технологии вам даже необязательно иметь COM-объекты на вашей машине: они могут быть получены с других машин в сети.

Резюме

В этой главе были описаны некоторые новые технологии и концепции. COM не так проста для понимания, и придется много потрудиться, чтобы действительно хорошо ее освоить. Однако использовать COM в десятки раз проще, чем понять ее, и это вы увидите в следующей главе. Здесь вы также познакомились с DirectX и всеми его компонентами. Как только в следующей главе вы побольше узнаете о каждом из компонентов DirectX и о том, как их использовать, вы сможете убедиться на практике, насколько хорошо и слаженно они работают в игровых программах.

ГЛАВА 6

Первое знакомство с DirectDraw

В этой главе читателю предстоит первое знакомство с одним из наиболее важных компонентов DirectX — DirectDraw. Заложенная в DirectDraw технология обеспечивает работу двухмерной графики, а кроме того, этот компонент выполняет функции уровня буфера кадров, на котором строится изображение Direct3D. Таким образом, DirectDraw — это, возможно, наиболее мощная технология в составе DirectX. В DirectX версии 8.0 компонент DirectDraw полностью интегрирован в Direct3D. Однако понимания технологии DirectDraw более чем достаточно для создания любых графических приложений, какие только могли быть написаны для операционной системы DOS16/32. Технология DirectDraw — это ключ к пониманию ряда концепций технологии DirectX, поэтому не жалейте сил для ее освоения!

Ниже перечислены основные темы данной главы.

- Интерфейсы DirectDraw
- Создание объекта DirectDraw
- Взаимодействие с Windows
- Ознакомление с режимами работы
- О тонкостях работы с цветами
- Построение выводимой поверхности

Интерфейсы DirectDraw

DirectDraw состоит из ряда *интерфейсов* (interfaces). Как упоминалось в главе 5, "Основы DirectX и COM", где шла речь о созданной компанией Microsoft модели составных объектов (Component Object Model — COM), интерфейс — это не что иное, как набор функций и/или методов, с помощью которого осуществляется взаимодействие с компонентами.

Рассмотрим рис. 6.1, на котором проиллюстрированы интерфейсы DirectDraw. Следует иметь в виду, что описание интерфейсов приводится без упоминания их версий; так что пока обсуждение будет чисто абстрактным. Например, уже разработаны IDirectDraw всех версий вплоть до 7.0, поэтому при реальном использовании этого интерфейса речь идет о IDirectDraw7. Однако на данном этапе внимание уделяется основам работы интерфейсов и взаимоотношениям между ними.

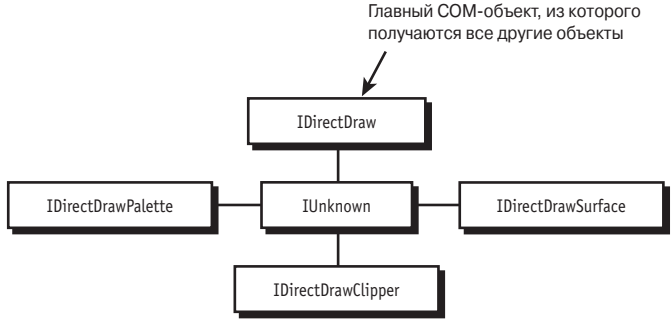


Рис. 6.1. Интерфейсы DirectDraw

Характеристики интерфейсов

Как видно из рис. 6.1, в состав DirectDraw входит всего пять интерфейсов, характеристики которых описаны ниже.

IUnknown. Из этого основного интерфейса должны порождаться все объекты COM, и DirectDraw — не исключение. Интерфейс IUnknown состоит из функций Addref(), Release(), QueryInterface(), переопределяемых каждым из остальных интерфейсов.

IDirectDraw. Этот интерфейс — основной объект, который необходимо создать, чтобы приступить к работе с DirectDraw. Интерфейс IDirectDraw буквальным образом представляет видеокарту и поддерживает аппаратное обеспечение компьютера. Интересно заметить, что теперь, когда в операционных системах Windows 98/ME/XP/NT/2000 предусмотрена поддержка нескольких мониторов (Multiple Monitor Support — MMS), в системе может быть установлено несколько видеокарт, а следовательно, иметься несколько объектов DirectDraw. Однако в настоящей книге предполагается, что в компьютере только одна видеокарта и для представления объектом DirectDraw всегда выбирается карта, установленная по умолчанию, даже если реально в системе есть несколькovидеокарт.

IDirectDrawSurface. Этот интерфейс отвечает за создание, преобразование и отображение с помощью DirectDraw текущей поверхности (или текущих поверхностей) дисплея. Поверхность DirectDraw может храниться как в памяти самой видеокарты (в *видеопамяти*), так и в памяти системы. Поверхности бывают двух видов: *первичные* (primary surfaces) и *вторичные* (secondary surfaces).

Первичные поверхности обычно представляют текущее содержимое видеобуфера, которое преобразуется видеокартой в растровый формат и отображается на дисплее. Вторичные поверхности обычно остаются за кадром. В большинстве случаев для представления текущего изображения на мониторе формируется одна первичная поверхность, а затем одна или несколько вторичных поверхностей, которые представляют растровое отображение объекта, и/или *задние буферы* (back buffers) для представления закадровых областей расположения рисунка, на которых формируется очередной кадр анимации. Подробности работы с поверхностями будут освещены в дальнейших разделах этой главы, а пока можете обратиться к графической схеме на рис. 6.2.

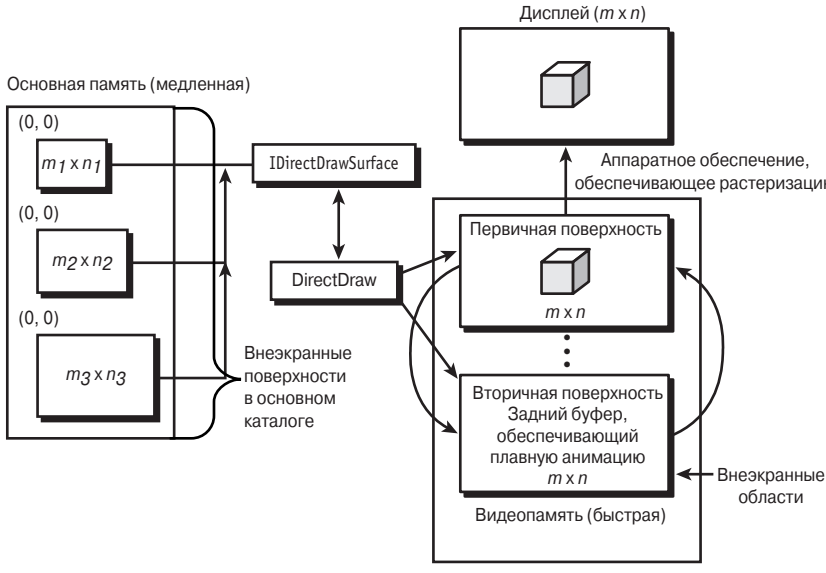


Рис. 6.2. Поверхности DirectDraw

IDirectDrawPalette. Технология DirectDraw оснащена средствами, предназначенными для всевозможных цветовых пространств — от однобитовых монохромных до 32-битовых Ultra-True Color. Таким образом, интерфейс IDirectDrawPalette поддерживается в DirectDraw для работы с цветовыми палитрами в режимах представления видеоизображений, в которых используется 256 или меньше цветов (в нескольких демонстрационных примерах этой главы интенсивно используется 256-цветный режим, потому что он быстрее всего преобразуется в растровый формат программными средствами). Одним словом, интерфейс IDirectDrawPalette применяется для создания, загрузки и преобразования палитр, а также для их присоединения к отображаемым поверхностям, в роли которых могут выступать первичные или вторичные поверхности, создаваемые для приложений DirectDraw. На рис. 6.3 приведена схема взаимоотношений между отображаемой поверхностью и палитрой DirectDraw.

IDirectDrawClipper. Этот интерфейс помогает выполнять отсечение графического изображения по границам области, а также операции по ограничению битового образа некоторым подмножеством, представляющим собой отображаемую на экране поверхность. В большинстве случаев отсекатели DirectDraw используются для оконных приложений DirectDraw и/или для усечения битового образа, чтобы ограничить его рамками отображаемой поверхности, как первичной, так и вторичной. К достоинствам интерфейса IDirectDrawClipper относится то, что он по возможности использует преимущества аппаратного ускорения, выполняя ресурсоемкие операции поэлементной (пиксель за пикселем) обработки или преобразования части изображения, которые обычно необходимы для усечения битовых образов до размеров экрана.

Прежде чем перейти к созданию объектов DirectDraw, напомним некоторые сведения из предыдущей главы, в которой речь шла о модели COM. DirectDraw, как и все другие компоненты DirectX, постоянно обновляются. Таким образом, хотя до сих пор в данной главе интерфейсы DirectDraw фигурировали под общими названиями — IDirectDraw, IDirectDrawSurface, IDirectDrawPalette и IDirectDrawClipper, по большей части эти интерфейсы уже обновлены и доступны их новые версии. Например, как уже упоминалось, в DirectX версии 7.0 интерфейс IDirectDraw обновлен до IDirectDraw7.

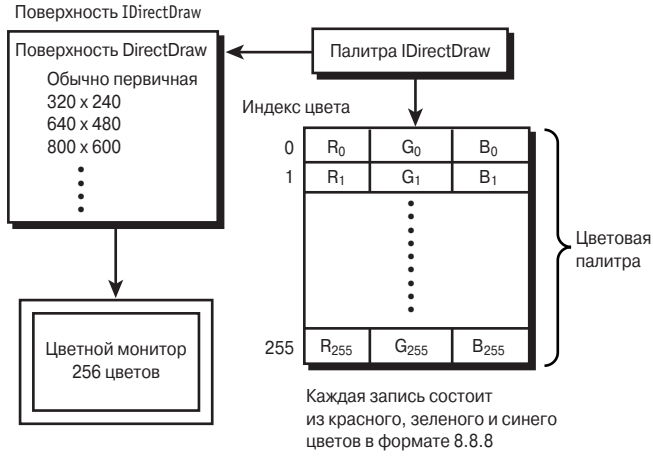


Рис. 6.3. Взаимосвязь между поверхностями и палитрами DirectDraw

Сказанное выше означает, что для использования преимуществ самого нового производительного программного и аппаратного обеспечения всегда следует выполнять запрос IUnknown::QueryInterface() для получения новейших версий интерфейсов. Однако, чтобы узнать об этом подробнее, вам придется заглянуть в документацию по DirectX SDK. В данной книге используется DirectX 8.0, но следует не забывать, что при обновлении до DirectX 9.0 возможно появление новых весьма полезных интерфейсов. Здесь же я постараюсь свести количество ненужной информации к минимуму. В большинстве случаев вам требуется лишь небольшая часть "побрякушек", которые появляются в новых версиях.

Совместное использование интерфейсов

Ниже кратко описан процесс создания приложения DirectDraw с помощью всех упомянутых интерфейсов.

1. Создайте основной объект DirectDraw и с помощью функции QueryInterface() получите интерфейс IDirectDraw7 (либо создайте его непосредственно с помощью функции DirectDrawCreateEx()). Пользуясь этим интерфейсом, установите уровень совместного доступа (cooperation level) и видеорежим.
2. С помощью интерфейса IDirectDrawSurface7 создайте, как минимум, первичную поверхность, которая будет выводиться на экран. Если видеорежим использует не более 8 бит на пиксель, вам потребуется определить палитру.
3. С помощью интерфейса IDirectDrawPalette создайте палитру, инициализируйте ее тройками чисел в формате RGB и подключите к интересующей вас поверхности.
4. Если приложение DirectDraw планируется запускать в оконном режиме или если битовые образы могут иногда выходить за рамки видимой поверхности DirectDraw, создайте хотя бы один отсекатель (clipper). Его размеры подбираются в соответствии с размерами видимой области окна (рис. 6.4).
5. Отобразите первичную поверхность.

Множество деталей в этом описании, конечно же, опущены, однако главное — суть процесса использования различных интерфейсов. Имея это в виду, рассмотрим описанный выше процесс подробнее и заставим работать все эти интерфейсы.

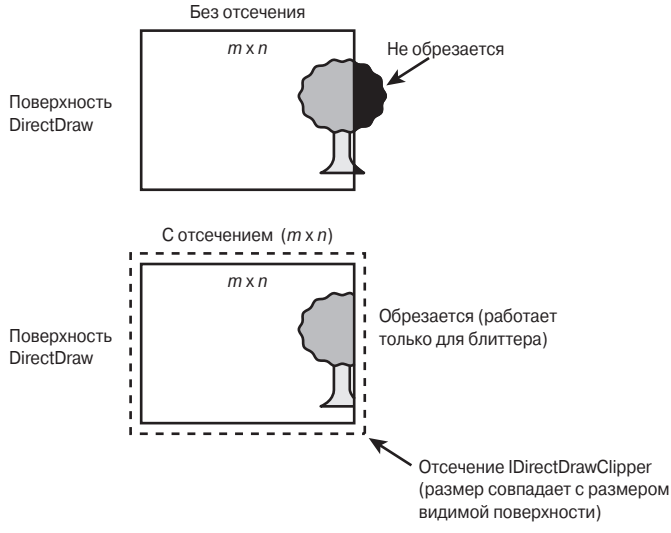


Рис. 6.4. Отсекатели DirectDraw

СОВЕТ

При чтении оставшейся части главы полезно держать открытым заголовочный файл DDRAW.H и справочную систему DirectX SDK.

Создание объекта DirectDraw

В языке программирования C++ объект DirectDraw 1.0 создается с помощью функции DirectDrawCreate().

```
HRESULT WINAPI DirectDrawCreate(
  GUID FAR       *lpGUID,         // GUID объекта
  LPDIRECTDRAW FAR *lplpDD,    // приемник интерфейса
  IUnknown FAR    *pUnkOuter); // элемент COM
```

Здесь lpGUID — это *глобально уникальный идентификатор* (Globally Unique Identifier — GUID) драйвера того дисплея, который будет использоваться. В большинстве случаев достаточно задать значение NULL, что соответствует аппаратному обеспечению, которое выбирается по умолчанию.

lplpDD — это указатель на указатель, в котором в случае успешного завершения функции хранится указатель на интерфейс IDirectDraw. Обратите внимание, что функция возвращает интерфейс IDirectDraw, а *не* IDirectDraw7!

pUnkOuter — параметр для опытных программистов; мы же всегда будем использовать значение NULL.

Ниже приведен фрагмент кода, в котором вызов функции создает объект DirectDraw по умолчанию.

```
LPDIRECTDRAW lpdd = NULL; // Переменная для хранения
                         // указателя на IDirectDraw

// создание объекта DirectDraw
DirectDrawCreate(NULL, &lpdd, NULL);
```

В случае успешного вызова функции параметр lpdd будет полноценным интерфейсом IDirectDraw 1.0 объекта, но нас интересует интерфейс самой последней версии, т.е. IDirectDraw7. Однако, прежде чем научиться его получать, ознакомимся с обработкой ошибок.

Обработка ошибок в DirectDraw

Принципы обработки ошибок в DirectDraw легко понять. Имеется несколько макросов, проверяющих успешность работы каждой функции. Предоставленный компанией Microsoft способ проверки наличия ошибок, которые могут возникнуть при вызове функций DirectX, заключается в использовании двух макросов:
FAILED() — проверка наличия сбоя;
SUCCEEDED() — проверка успешного выполнения.

Имея такую информацию, добавим в программу код для обработки ошибок:

```
if (FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
{
  // ошибка
} // if
```

Аналогично можно проверить и успешность выполнения функции:

```
if (SUCCEEDED(DirectDrawCreate(NULL, &lpdd, NULL)))
{
  // переход к следующему шагу
} // if
else
{
  // ошибка
} // else
```

Лично я предпочитаю пользоваться макросом FAILED(); в этом случае логическая структура if-else состоит только из одной (первой) части и всегда есть возможность получить информацию о сбое в программе. Единственная проблема, связанная с использованием этих макросов, заключается в том, что они выдают очень скудную информацию; эти макросы предназначены для общего выявления проблемы, а не для точной диагностики. Чтобы узнать, в чем именно состоит проблема, всегда можно взглянуть на код возврата функции. В табл. 6.1 приведен список возможных кодов возврата функции DirectDrawCreate(), входящей в состав DirectX версии 6.0.

Таблица 6.1. Коды возврата функции DirectDrawCreate ()

Код возврата	*Описание*
DD_OK	Успешное выполнение
DDERR_DIRECTDRAWALREADYCREATED	Объект DirectDraw уже создан
DDERR_GENERIC	DirectDraw не в состоянии выяснить природу ошибки
DDERR_INVALIDDIRECTDRAWGUID	Данный глобально уникальный идентификатор устройства неизвестен
DDERR_INVALIDPARAMS	Неправильно заданы параметры
DDERR_NODIRECTDRAWHW	Отсутствует аппаратное обеспечение
DDERR_OUTOFMEMORY	Нехватка памяти

К сожалению, при использовании этих констант в условных операторах нельзя полагаться на то, что Microsoft оставит в последующих версиях DirectX приведенные выше коды возврата без изменений. Однако будем считать, что такой программный код будет работать достаточно надежно.

```
if (DirectDrawCreate(...)!=DD_OK)
{
  // ошибка
} // if
```

Более того, код возврата DD_OK определен для всех функций DirectDraw, поэтому его можно уверенно использовать в программах.

Обновление версии интерфейса

Как уже отмечалось, вы можете пользоваться базовым интерфейсом IDirectDraw, который хранится в lpdd после вызова функции DirectDrawCreate(). Однако вы можете также обновить интерфейс до последней версии, запросив новый интерфейс с помощью метода QueryInterface(), который имеется в интерфейсе IUnknown и является частью реализации каждого интерфейса DirectDraw. Самый последний интерфейс DirectDraw в составе DirectX версии 7.0 — это IDirectDraw7. Указатель на него можно получить с помощью следующего кода:

```
LPDIRECTDRAW lpdd = NULL;   // Стандартный DirectDraw 1.0
LPDIRECTDRAW lpdd7 = NULL;  // Интерфейс DirectDraw 7.0

// Сначала создаем базовый интерфейс IDirectDraw
if (FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
{
  // ошибка
} // if

// запрос интерфейса IDirectDraw7
if (FAILED(lpdd->QueryInterface(IID_IDirectDraw7,
                  (LPVOID *)&lpdd7)))
{
  // ошибка
} // if
```

Обратите внимание на такие важные особенности приведенного кода:

- способ вызова функции QueryInterface();
- константу IID_IDirectDraw7, с помощью которой запрашивается интерфейс IDirectDraw7.

Вообще говоря, все вызовы интерфейсов имеют вид

```
указатель_интерфейса->метод(параметры...);
```

Кроме того, все идентификаторы интерфейсов имеют вид IID_IDirectCD, где C обозначает конкретный компонент (например, для DirectDraw вместо C нужно подставить Draw, для DirectSound — Sound, для DirectInput — Input и т.д.), а D — номер, который изменяется от 2 до *n* и указывает версию интерфейса. Все эти константы можно найти в файле DDRAW.H.

Возвращаясь к нашему примеру, мы видим, что возникла небольшая дилемма: в наличии имеется как интерфейс IDirectDraw, так и интерфейс IDirectDraw7. Что же делать в такой ситуации? От старого интерфейса лучше избавиться, ведь он нам больше не нужен. Делается это так:

```
lpdd->Release();
lpdd = NULL; // для надежности присвоим ему значение NULL
```

Начиная с этого момента, все вызовы методов будут выполняться с помощью нового интерфейса IDirectDraw7.

ВНИМАНИЕ

Использование новых возможностей интерфейса IDirectDraw7 связано с небольшой подготовительной работой и повышенной ответственностью. Проблема не только в том, что интерфейс IDirectDraw7 более сложен и улучшен по сравнению с базовым. Дело в том, что во многих случаях он требует и возвращает новые структуры данных, которые не определены в DirectX 1.0. Единственный способ убедиться в том, что вы используете верные структуры данных, — заглянуть в документацию DirectX SDK.

Кроме использования базового интерфейса IDirectDraw и функции QueryInterface() имеется и более прямой способ получения интерфейса IDirectDraw7. В модели COM можно получить указатель на любой интерфейс; для этого достаточно знать *идентификатор интерфейса* (Interface ID — IID), который представляет интересующий нас интерфейс. В большинстве случаев я предпочитаю не пользоваться низкоуровневыми функциями COM, потому что жизнь и так полна сложностей и неожиданностей. Однако при работе с компонентом DirectMusic без таких низкоуровневых элементов модели COM никак не обойтись, так что нам пора хотя бы ознакомиться с этим процессом. Вот как можно непосредственно получить интерфейс IDirectDraw7:

```
// Инициализируем COM, что приводит к загрузке необходимых
// библиотек COM, если они все еще не загружены
if (FAILED(CoInitialize(NULL)))
{
  // ошибка
} // if

// С помощью функции CoCreateInstance()
// создаем объект DirectDraw
if (FAILED(CoCreateInstance(&CLSID_DirectDraw
                NULL,
                CLSCTX_ALL,
                &IID_IDirectDraw7,
                &lpdd7)))
{
  // ошибка
} // if

// Перед использованием объекта DirectDraw его необходимо
// инициализировать с помощью метода Initialize
if (FAILED(IDirectDraw7_Initialize(lpdd7, NULL)))
{
  // ошибка
} // if

// Поскольку COM больше не нужна, деинициализируем ее
CoUninitialize();
```

Приведенный выше код представляет собой способ, которым компания Microsoft рекомендует создавать объект DirectDraw. Однако в этом методе есть одна хитрость — использование макроса

```
IDirectDraw7_Initialize(lpdd7, NULL);
```

Без этого вполне можно обойтись, заменив приведенную выше строку следующей:

```
lpdd7->Initialize(NULL);
```

Это позволит полностью оставаться в рамках COM. Как в первой, так и во второй строках инициализации значение NULL относится к видеоустройству, в роли которого в данном случае выступает установленный по умолчанию драйвер (именно поэтому параметр остается равным значению NULL). В любом случае приведенная последней строка кода позволяет легко понять, какие инструкции содержатся в макросе, а понимание всегда облегчает жизнь. А теперь хорошая новость: компания Microsoft реализовала функцию, которая позволяет создать интерфейс IDirectDraw7 без промежуточных этапов. Раньше этот способ обычно не использовался, однако в DirectX версии 7.0 появилась функция под названием DirectDrawCreateEx(), которая уже упоминалась в предыдущей главе. Прототип этой функции имеет следующий вид:

```
HRESULT WINAPI DirectDrawCreateEx(
 GUID FAR *lpGUID,           // GUID драйвера, для активного дисплея
                             // используется значение NULL
 LPVOID  *lplpDD,            // Переменная для хранения интерфейса
 REFIID   iid,               // идентификатор запрашиваемого интерфейса
 IUnknown FAR *pUnkOuter // У нас - всегда NULL
);
```

Функция DirectDrawCreateEx() очень похожа на более простую функцию DirectDrawCreate(), но ей требуется больше параметров, и она позволяет создавать любой интерфейс DirectDraw. Вызов функции DirectDrawCreateEx(), позволяющий создать интерфейс IDirectDraw7, выглядит следующим образом:

```
LPDIRECTDRAW7 lpdd; // версия 7.0

// создание объекта интерфейса DirectDraw версии 7.0
DirectDrawCreateEx(NULL, (void **)&lpdd, IID_IDirectDraw7, NULL);
```

Единственная хитрость здесь состоит в преобразовании указателя интерфейса к типу (void**) и в том, какое значение для запрашиваемого интерфейса необходимо передать параметру iid. В остальном все очень просто, причем нормально работает!

Подведем итог. Чтобы работать с компонентом DirectDraw последней версии (а именно 7.0, так как Microsoft выпустила DirectX версии 8.0 без DirectDraw), нужен интерфейс IDirectDraw7. Для этого можно воспользоваться базовой функцией DirectDrawCreate(), чтобы получить интерфейс IDirectDraw версии 1.0, а затем с помощью функции QueryInterface() запросить интерфейс IDirectDraw7. Есть и другие способы: можно воспользоваться низкоуровневыми функциями COM для непосредственного получения интерфейса IDirectDraw7 или функцией DirectDrawCreateEx(), которая появилась вместе с выпуском DirectX 7.0. Неплохо, не так ли? DirectX начинает напоминать систему X Windows, в которой каждое действие можно выполнить тысячами разных способов.

Теперь, научившись создавать объект DirectDraw и получать самый новый интерфейс, перейдем к следующему этапу нашего плана по освоению DirectDraw.

Взаимодействие с Windows

Как известно, операционная система Windows — это совместно используемая среда (по крайней мере такова идея, лежащая в ее основе). Однако программисту нужно знать, как заставить эту операционную систему взаимодействовать с создаваемым им кодом, а об этом еще ничего не было сказано. Так или иначе, но DirectX аналогична любой Win32-

совместимой системе, и приложение DirectX должно, как минимум, информировать Windows о намерении использовать различные ресурсы, чтобы другие приложения Windows не пытались запрашивать (и не получали в свое распоряжение) те ресурсы, которые контролируются приложением DirectX. По сути, DirectX может получить управление всеми ресурсами и сохранять его до тех пор, пока Windows предоставляется информация о выполняемых с этими ресурсами операциях.

В случае использования DirectDraw практически единственная вещь, в которой заинтересован программист, — это аппаратное обеспечение монитора. Необходимо различать два режима работы приложений: полноэкранный и оконный.

В *полноэкранном* режиме (full-screen mode) работа DirectDraw во многом похожа на работу старой DOS-программы: игре выделяется вся поверхность экрана, и запись информации ведется непосредственно в память видеоустройств. Никакое другое приложение не может вмешиваться в работу этих устройств. *Оконный режим* (windowed mode) несколько отличается от полноэкранного. В этом режиме компонент DirectDraw должен намного интенсивнее взаимодействовать с Windows, потому что другим приложениям может понадобиться обновить изображение в области своих клиентских окон (которые могут быть видимы пользователям). Поэтому в оконном режиме возможности контроля и монополизации видеоустройств существенно ограничены. При этом у нас остается возможность в полной мере использовать 2D- и 3D-ускорение. Однако обо всем по порядку.

В главе 7, "Постигаем секреты DirectDraw и растровой графики", большое внимание уделяется приложениям DirectX, работающим в оконном режиме, однако управлять ими несколько сложнее. Бо́льшая же часть данной главы посвящена полноэкранному режиму, так как с ним легче работать.

Теперь, когда вы получили некоторое представление о причинах необходимости взаимодействия между Windows и DirectX, рассмотрим, как сообщить операционной системе Windows о том, что ваша программа намерена поработать с ней. Уровень совместного доступа DirectDraw устанавливается с помощью функции IDirectDraw7::SetCooperativeLevel(), которая является методом интерфейса IDirectDraw7.

C++

Для программистов, которые работают с языком C, синтаксис инструкции IDirectDraw7::SetCooperativeLevel() может оказаться несколько непонятным. Оператор :: называется *оператором разрешения области видимости* (scope resolution operator), и приведенный синтаксис попросту означает, что функция SetCooperativeLevel() — это метод (или функция-член) интерфейса IDirectDraw7. По сути, интерфейс — это класс, который представляет собой не что иное, как структуру с данными и таблицей виртуальных функций. В некоторых случаях название интерфейса в префиксе метода будет опускаться и метод будет записываться просто как SetCooperativeLevel(). Однако следует помнить, что все функции DirectX являются частью какого-то интерфейса и поэтому их нужно вызывать с помощью указателя: lpdd->function(...).

Приведем прототип функции IDirectDraw7::SetCooperativeLevel().

```
HRESULT SetCooperativeLevel(
    HWND hWnd,      // дескриптор окна
    DWORD dwFlags); // управляющие флаги
```

В случае успешного завершения эта функция возвращает значение DD_OK, а в противном случае — код ошибки.

Интересно, что здесь в формуле DirectX первый раз фигурирует дескриптор окна (вы просто используете в качестве этого параметра дескриптор вашего главного окна).

Второй, и последний, параметр функции SetCooperativeLevel() — управляющие флаги dwFlags, которые непосредственно влияют на способ работы DirectDraw с операционной системой Windows. В табл. 6.2 приведен список наиболее распространенных значений флагов, которые могут комбинироваться с использованием побитового оператора OR.

Таблица 6.2. Управляющие флаги функции SetCooperativeLevel()

Значение	*Описание*
DDSCL_ALLOWMODEX	Позволяет использовать режимы визуального отображения Mode X (320×200,240,400). Этот флаг можно использовать только при наличии флагов DDSCL_EXCLUSIVE и DDSCL_FULLSCREEN
DDSCL_ALLOWREBOOT	Позволяет операционной системе выявлять нажатие комбинации клавиш <Ctrl+Alt+Del> в исключительном (полноэкранном) режиме
DDSCL_EXCLUSIVE	Запрашивает исключительный режим. Этот флаг нужно использовать совместно с флагом DDSCL_FULLSCREEN
DDSCL_FPUSETUP	Флаг указывает на то, что вызывающее приложение будет использовать сопроцессор для достижения оптимальной производительности Direct3D. Более подробную информацию об этом можно найти в разделе "DirectDraw Cooperative Levels and FPU Precision" документации DirectX SDK
DDSCL_FULLSCREEN	Флаг указывает на то, что будет использоваться полноэкранный режим. Другие приложения не смогут при этом выводить изображение на экран. Этот флаг необходимо использовать совместно с флагом DDSCL_EXCLUSIVE
DDSCL_MULTITHREADED	Запрашивает безопасную работу DirectDraw в многопоточном режиме. Это значение управляющего флага нас пока не интересует
DDSCL_NORMAL	Указывает, что приложение будет функционировать как обычное приложение Windows. Этот флаг нельзя использовать вместе с флагами DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE или DDSCL_FULLSCREEN
DDSCL_NOWINDOWCHANGES	Флаг указывает, что при активизации приложения DirectDraw окно этого приложения нельзя сворачивать или разворачивать

Внимательно просмотрев список значений флагов, можно прийти к выводу, что некоторые из них являются излишними (очень правильный вывод). По сути, флаги DDSCL_FULLSCREEN и DDSCL_EXCLUSIVE необходимо использовать совместно, и, если принято решение использовать один из режимов Mode X, флаги DDSCL_FULLSCREEN, DDSCL_EXCLUSIVE и DDSCL_ALLOWMODEX тоже должны применяться вместе. О действии остальных флагов можно догадаться по их названиям. В большинстве случаев полноэкранное приложение настраивается так:

```
lpdd7->SetCooperativeLevel(hwnd,
                DDSCL_FULLSCREEN|
                DDSCL_ALLOWMODEX|
                DDSCL_EXCLUSIVE|
                DDSCL_ALLOWREBOOT);
```

Обычное же приложение, работающее в оконном режиме, — следующим образом:

```
lpdd7->SetCooperativeLevel(hwnd, DDSCL_NORMAL);
```

При чтении последующих глав, в которых в частности идет речь о методах многопоточного программирования, может возникнуть желание добавлять флаг DDSCL_MULTITHREADED, чтобы обеспечить надежную работу игры.

Рассмотрим фрагмент кода, создающий объект DirectDraw и устанавливающий уровень совместного доступа.

```
LPDIRECTDRAW lpdd = NULL;  // Обычный DirectDraw 1.0
LPDIRECTDRAW7 lpdd7 = NULL; // Интерфейс DirectDraw 7.0

// Создаем базовый интерфейс IDirectDraw
if (FAILED(DirectDrawCreateEx(NULL, (void**)&lpdd7,
                  IID_IDirectDraw7, NULL);
{
  // ошибка
} // if

// Задаем уровень совместного доступа для приложения
// DirectDraw в оконном режиме, поскольку пока что
// вывод изображения на экран не предполагается
if (FAILED(lpdd7->SetCooperativeLevel(hwnd, DDSCL_NORMAL)))
{
  // ошибка
} // if
```

НА ЗАМЕТКУ

Иногда в целях экономии места вызов функции FAILED() и/или SUCCEEDED(), предназначенный для обработки ошибок, будет опускаться, но помнить о необходимости проверки наличия ошибок следует всегда!

Теперь у нас достаточно информации для создания работающего приложения DirectX, которое создает окно, запускает DirectDraw и задает уровень совместного доступа. Хотя мы еще не знаем, как выводить изображение, определенный начальный этап уже пройден. В качестве примера рассмотрим содержащуюся на прилагаемом компакт-диске демонстрационную программу DEMO6_1.CPP. При запуске этой программы на экране появится окно, подобное изображенному на рис. 6.5. Основой для рассматриваемой программы служит все тот же шаблон T3D Game Console, поэтому все изменения в функциях Game_Init() и Game_Shutdown() сводятся к созданию и установке уровня совместного доступа для DirectDraw.

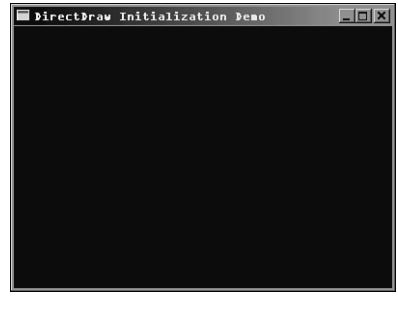


Рис. 6.5. Программа DEMO6_1.EXE в действии

Ниже приводится код этих функций из файла DEMO6_1.CPP. Как видите, настройка DirectDraw — дело несложное.

```
int Game_Init(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз после создания окна
// и перед входом в главный цикл обработки сообщений. Здесь
// выполняются все инициализирующие действия.

// Создаем базовый интерфейс IDirectDraw
if (FAILED(DirectDrawCreateEx(NULL, (void **)&lpdd,
                   IID_IDirectDraw7, NULL)))
{
  // ошибка
  return(0);
} // if

// Задаем для уровня совместного доступа флаг DDSCL_NORMAL,
// поскольку приложение будет работать в оконном режиме
lpdd->SetCooperativeLevel(main_window_handle, DDSCL_NORMAL);

// Успешное завершение, неудача или ваш код ошибки
return(1);

} // Game_Init

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

int Game_Shutdown(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается при завершении работы игры.
// В ней выполняются все действия по освобождению
// захваченных ресурсов и т.п.

// Освобождаем интерфейс IDirectDraw
if (lpdd)
{
  lpdd->Release();
  lpdd = NULL;
} // if

// Успешное завершение, неудача или ваш код ошибки
return(1);

} // Game_Shutdown
```

СОВЕТ

Если вы уже готовы заняться компиляцией исходного файла `DEMO6_1.CPP`, не забудьте, пожалуйста, вручную подключить библиотеку `DDRAW.LIB`, которая находится в каталоге DirectX 8.0 SDK `LIB\`, а также указать пути к заголовочным файлам DirectX в настройках вашего компилятора. Кроме того, следует указать, что вы создаете Win32. Мне очень надоело ежедневно получать не менее десятка писем от начинающих программистов, забывающих включить библиотечные файлы...

Установка режима

Следующий этап настройки DirectDraw, пожалуй, самый интересный. Обычно в DOS используются видеорежимы, поддерживаемые ROM BIOS. Однако в Windows это почти

невозможно из-за значительных изменений, которые должны быть выполнены в операционной системе при переключении видеорежима. При использовании DirectX все происходит проще и быстрее. Одна из основных задач DirectDraw состоит в том, чтобы сделать переключение режима понятным для программиста. Вам больше не нужно самостоятельно программировать работу регистров управления VGA/CRT — достаточно просто вызвать соответствующую функцию, и будет установлен любой нужный видеорежим (конечно же, если это позволяет видеокарта).

Функция, предназначенная для установки видеорежима, называется SetDisplayMode() и является методом интерфейса IDirectDraw7; пользуясь синтаксисом языка C++, это можно записать следующим образом: IDirectDraw7::SetDisplayMode(). Ниже приведен прототип этой функции.

```
HRESULT SetDisplayMode(
  DWORD dwWidth,        // Ширина режима в пикселях
  DWORD dwHeight,       // Высота режима в пикселях
  DWORD dwBPP,          // Количество битов на пиксель
  DWORD dwRefreshRate, // Частота обновления (0 - частота по умолчанию)
  DWORD dwFlags);       // дополнительные флаги (по умолчанию 0)
```

Как обычно, при успешном завершении функция возвращает значение DD_OK.

Ну что же, можно сказать лишь одно: "Это слишком хорошо!" Вы когда-нибудь пробовали самостоятельно установить какой-нибудь видеорежим типа 320×400 или 800×600? Пользуясь рассматриваемой функцией DirectDraw, достаточно указать ширину и высоту экрана и требуемую глубину цвета — и все! DirectDraw сам справится со всей кропотливой работой по настройке видеокарты, а по возможности — и частоты обновления. Более того, в этом режиме гарантируется линейность буфера видеопамяти... но об этом позже. Взгляните на табл. 6.3, в которой приведен краткий список наиболее часто используемых видеорежимов, а также соответствующие им значения глубины цвета.

Таблица 6.3. Параметры наиболее распространенных видеорежимов

Ширина	*Высота*	*Глубина*	*Mode X*
320	200	8	*
320	240	8	*
320	400	8	*
512	512	8, 16, 24, 32	
640	480	8, 16, 24, 32	
800	600	8, 16, 24, 32	
1024	768	8, 16, 24, 32	
1280	1024	8, 16, 24, 32	

Примечание. Многие видеокарты позволяют установить режимы с более высоким разрешением.

Интересно, что можно запросить любой видеорежим, какой только пожелаешь. Например, можно выбрать разрешение 400×400, и, если видеодрайвер способен его установить, он это сделает. Однако все же лучше придерживаться режимов, перечисленных в табл. 6.3, так как это наиболее распространенные стандартные видеорежимы.

СЕКРЕТ

В составе Win32 API имеется функция, позволяющая установить видеорежим, который использовался ранее, однако эта функция только вносит хаос в систему и все запутывает.

Вернемся к функции SetDisplayMode(). Смысл первых трех ее параметров очевиден, а остальные два нуждаются в пояснении. Параметр dwRefreshRate используется для изменения установленной по умолчанию частоты обновления. Например, пусть разрешению 320×200 по умолчанию соответствует частота 70 Гц; с помощью же этого параметра при желании можно установить ее равной 60 Гц. Но пока оставим частоту обновления в покое и сосредоточимся на других темах. Для этого значение параметра dwRefreshRate задается равным 0 (при этом драйвер использует частоту, заданную по умолчанию).

Последний параметр dwFlags — это еще один всеобъемлющий и малополезный флаг типа WORD. В данном случае он используется в качестве переключателя, с помощью которого можно установить режим VGA 13h для разрешения 320×200 вместо режима Mode X для этого же разрешения; это делается с помощью флага DDSDM_STANDARDVGAMODE. Однако об этом параметре, как и о предыдущем, не стоит беспокоиться. Если создается игра, в которой используется разрешение 320×200, конечно, можно поэкспериментировать с данным флагом, по очереди устанавливая режимы VGA 13h и Mode X для разрешения 320×200, чтобы узнать, какой из них быстрее. Однако вы увидите, что разницей в производительности можно пренебречь. Так что пока значение параметра dwFlags будем задавать равным 0.

Но довольно предисловий — приступим к делу. Для этого необходимо создать объект DirectDraw, установить уровень совместного доступа и, наконец, задать видеорежим:

```
lpdd->SetDisplayMode(width, height, bpp, 0, 0);
```

Например, чтобы установить режим с разрешением 640×480 и 256 цветами (8 бит), воспользуйтесь такой инструкцией:

```
lpdd->SetDisplayMode(640, 480, 8, 0, 0);
```

Режим с разрешением 800×600 и 16-битовыми цветами можно установить так:

```
lpdd->SetDisplayMode(800, 600, 16, 0, 0);
```

Разница между двумя приведенными выше режимами намного глубже, чем простое различие разрешений; она связана еще и с глубиной цвета (color depth). Работа в 8-битовом режиме полностью отличается от работы в 16-, 24- и 32-битовом режимах. Напомним, что подробно использование палитр описано в главах 3, “Профессиональное программирование в Windows”, и 4, “GDI, управляющие элементы и прочее”; все сказанное там остается в силе и при программировании в DirectDraw. Значит, устанавливая 8-битовый режим, программист должен создать палитру и заполнить ее записями в формате 8.8.8 RGB.

При работе в режиме с глубиной цвета 16, 24 или 32 бит на пиксель об этом шаге можно не беспокоиться. Кодированные RGB-данные можно записывать прямо в видеобуфер (если вы, конечно, знаете, как это делается). И все же научиться работать с палитрами DirectDraw необходимо (это будет наша следующая тема). Однако, прежде чем перейти к палитрам, рассмотрим завершенный пример по созданию полноэкранного приложения DirectX с разрешением 640×480×8.

Программа, которая выполняет указанные выше действия, содержится на прилагаемом компакт-диске в файле с названием DEMO6_2.CPP. Здесь можно было бы привести рисунок, показывающий, как работает данная программа, однако на нем изображен лишь черный прямоугольник, потому что пример демонстрирует создание полноэкранного приложения. Но код, благодаря которому достигается этот эффект, будет показан обязательно. Пример, как обычно, основан на шаблоне Game Console, который соответствующим образом модифицирован; кроме того, как видно из приведенного ниже листинга файла DEMO6_2.CPP, в разделы Game_Init() и Game_Shutdown() внесены изменения, связанные с DirectX. Рассмотрим листинг и поразимся его простоте...

```
int Game_Init(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз после создания окна
// и перед входом в главный цикл обработки сообщений. Здесь
// выполняются все инициализирующие действия.

// сначала создаем базовый интерфейс IDirectDraw
if (FAILED(DirectDrawCreateEx(NULL, (void **)&lpdd,
        IID_IDirectDraw7, NULL)))
{
   // Ошибка
   return(0);
} // if

// Работа в полноэкранном режиме
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle,
        DDSCL_FULLSCREEN| DDSCL_ALLOWMODE X |
        DDSCL_EXCLUSIVE| DDSCL_ALLOWREBOOT)))
{
   // Ошибка
   return(0);
} // if

// Установка видеорежима 640x480x8
if (FAILED(lpdd->SetDisplayMode(SCREEN_WIDTH,
        SCREEN_HEIGHT, SCREEN_BPP,0,0)))
{
   // Ошибка
   return(0);
} // if

// Успешное завершение, неудача или ваш код ошибки
return(1);

} // Game_Init

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

int Game_Shutdown(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается при завершении работы игры.
// В ней выполняются все действия по освобождению
// захваченных ресурсов и т.п.

// Освобождаем интерфейс IDirectDraw
if (lpdd)
{
   lpdd->Release();
   lpdd = NULL;
} // if

// Успешное завершение, неудача или ваш код ошибки
return(1);

} // Game_Shutdown
```

Пока здесь все еще не хватает двух вещей: управления палитрой (в режимах с 256 цветами) и доступа к дисплейным буферам. Сначала рассмотрим проблемы, связанные с цветом.

О тонкостях работы с цветом

Технология DirectDraw поддерживает несколько различных глубин цветов, включая 1, 2, 4, 8, 16, 24 и 32 бит на пиксель. Очевидно, глубины 1, 2 и 4 бит на пиксель несколько устарели, поэтому не будем их рассматривать. Основной интерес представляют глубины 8, 16, 24 и 32 бит на пиксель. Большинство игр работают либо в 8-битовом режиме с палитрой (в этом режиме программа работает очень быстро; кроме того, он хорошо подходит для учебных целей), либо в 16- или 24-битовом режиме с использованием цветов RGB. Как видно из рис. 6.6, режимы RGB в процессе работы записывают в буфер кадров значения типа WORD одинакового размера. Режим с палитрой работает с использованием таблицы поиска цветов, в которой для каждого байта из буфера кадров имеется запись, указывающая RGB-цвет. Таким образом, всего имеется 256 различных значений цветов.

Все, что вам нужно научиться делать, — это создавать 256-цветную палитру и сообщать DirectDraw о намерении ее использовать. Это делается следующим образом.

1. Создайте одну или несколько структур данных для палитр в виде массива, состоящего из 256 значений типа PALETTENTRY.
2. Создайте на основе объекта DirectDraw объект интерфейса IDirectDrawPalette палитры DirectDraw. Во многих случаях этот объект будет непосредственно отображаться в регистры устройства VGA, связанные с палитрой.
3. Присоедините палитру к поверхности, чтобы все представляемые данные отображались нужными цветами.
4. (*Необязательно*) При желании палитру или отдельные ее записи можно изменять. Этот шаг понадобится в том случае, если первый этап опущен, а на втором происходит передача нулевой палитры. При создании интерфейса палитры сами цвета этой палитры могут быть определены немедленно, но это можно сделать и позже. Таким образом, второй шаг может оказаться первым, если программист не забывает о необходимости заполнить палитру на последующих этапах.

Приступим к созданию структуры данных палитры. Структура данных палитры — это не что иное, как массив, состоящий из 256 записей палитры, основанных на приведенной ниже структуре Win32 типа PALETTEENTRY.

```
typedef struct tagPALETTEENTRY
{
  BYTE peRed;   // Красный компонент, 8-бит
  BYTE peGreen; // Зеленый компонент, 8-бит
  BYTE peBlue;  // Синий компонент, 8-бит
  BYTE peFlags; // Управляющие флаги: PC_NOCOLLAPSE
} PALETTEENTRY;
```

Уже знакомо? Тем лучше! Итак, для создания палитры достаточно просто создать массив таких структур:

```
PALETTEENTRY palette[256];
```

Затем созданный массив заполняется любым приемлемым для вас способом. При этом следует соблюдать лишь одно правило: в поле peFlags необходимо установить флаг PC_NOCOLLAPSE. Это требуется для того, чтобы палитра не оптимизировалась

Win32/DirectX. Учитывая сказанное, приведем пример создания палитры из случайных цветов, в которой нулевой цвет — черный, а цвет номер 255 — белый.

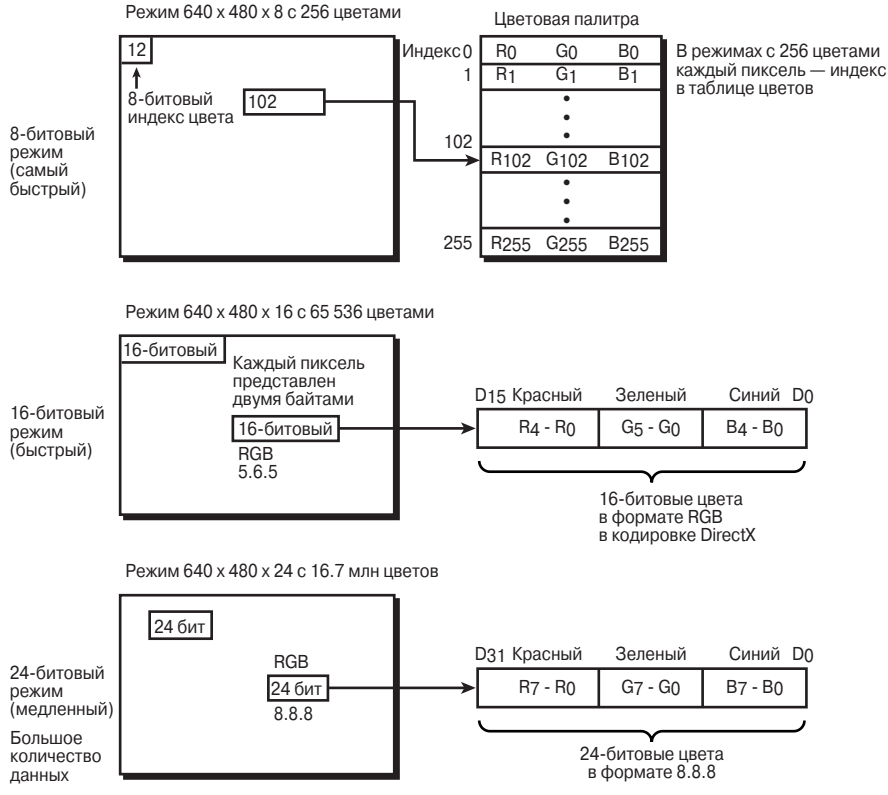


Рис. 6.6. Сравнение различных глубин цветов

```
PALETTEENTRY palette[256]; // Хранилище палитры

// Заполняем палитру цветами
for (int color=1; color < 255; color++)
{
   // Заполнение случайными значениями RGB
   palette[color].peRed   = rand()%256;
   palette[color].peGreen = rand()%256;
   palette[color].peBlue  = rand()%256;

   // Установка флага PC_NOCOLLAPSE
   palette[color].peFlags = PC_NOCOLLAPSE;
} // for color

// Заполнение элементов 0 и 255 черным и белым цветами
palette[0].peRed = 0;
palette[0].peGreen = 0;
palette[0].peBlue = 0;
palette[0].peFlags = PC_NOCOLLAPSE;
```

```
palette[255].peRed = 255;
palette[255].peGreen = 255;
palette[255].peBlue = 255;
palette[255].peFlags = PC_NOCOLLAPSE;
```

Вот и все! Конечно же, можно создать несколько палитр и заполнить их разными цветами — все в ваших руках.

Перейдем к следующему шагу — созданию интерфейса IDirectDrawPalette. К счастью, этот интерфейс остался таким же, как в версии DirectX 6.0, поэтому нет нужды привлекать функцию QueryInterface() или что-то еще. Ниже приведен прототип функции IDirectDraw7::CreatePalette(), с помощью которой создается объект-палитра.

```
HRESULT CreatePalette(
  DWORD dwFlags,                  // Управляющие флаги
  LPPALETTEENTRY lpColorTable, // Данные палитры либо NULL
  LPDIRECTDRAWPALETTE FAR *lplpDDPalette,
                                  // Возвращаемый интерфейс
  IUnknown FAR *pUnkOuter);    // Используем здесь NULL
```

В случае успешного выполнения функция возвращает значение DD_OK.

Рассмотрим параметры функции. Первый из них, dwFlags, управляет различными свойствами палитры; вскоре мы поговорим о нем подробнее. Следующий параметр содержит указатель на исходную палитру или значение NULL, если палитра в функцию не передается. Далее идет указатель на переменную для хранения указателя на интерфейс IDirectDrawPalette, куда в случае успешного выполнения функции передается интерфейс. Наконец, pUnkOuter — это параметр для программистов, имеющих опыт работы с моделью COM; здесь мы всегда будем использовать значение NULL.

Единственным представляющим интерес параметром является, конечно же, параметр-флаг dwFlags. Рассмотрим подробнее предоставляемые им возможности. В табл. 6.4 перечислены значения, которые можно использовать для создания слова флагов с применением операции побитового OR.

Таблица 6.4. Управляющие флаги функции CreatePalette()

Значение	*Описание*
DDPCAPS_1BIT	1-битовый цвет. В таблице цветов содержится две записи
DDPCAPS_2BIT	2-битовый цвет. В таблице цветов содержится четыре записи
DDPCAPS_4BIT	4-битовый цвет. В таблице цветов содержится 16 записей
DDPCAPS_8BIT	8-битовый цвет. Самый употребляемый флаг. В таблице цветов содержится 256 записей
DDPCAPS_8BITENTRIES	Этот флаг предназначен для активизации *индексированной палитры* (indexed palettes) и применяется для 1-, 2- и 4-битовых палитр. Просто скажите “нет”
DDPCAPS_ALPHA	Указывает на то, что член peFlags ассоциированной структуры PALETTENTRY следует интерпретировать как единое 8-битовое альфа-значение, управляющее прозрачностью. Палитру, которая создается с помощью этого флага, можно присоединять только к текстуре, образованной с использованием флага DDSCAPS_TEXTURE. Флаг предназначен для опытных программистов

Окончание табл. 6.4

Значение	*Описание*
DDPCAPS_ALLOW256	Флаг указывает, что для данной палитры можно определить все 256 записей. Обычно записи 0 и 255 зарезервированы для черного и белого цветов соответственно, а в некоторых системах, например в NT, эти записи нельзя изменять ни при каких обстоятельствах. Однако в большинстве случаев этот флаг не нужен, поскольку запись 0 — это и так обычно черный цвет, а кроме того, в большинстве палитр запись 255 можно оставить за белым цветом. Все зависит от программиста
DDPCAPS_INITIALIZE	Инициализация палитры цветами, содержащимися в массиве цветов, который передается с помощью параметра lpDDColorArray. Применяется для загрузки данных в аппаратную палитру
DDPCAPS_PRIMARYSURFACE	Присоединение палитры к первичной поверхности. При этом изменение таблицы цветов этой палитры оказывает немедленное влияние на дисплей, если только не установлен флаг DDPCAPS_VSYNC
DDPCAPS_VSYNC	Установка флага приводит к тому, что обновление палитры происходит только во время обратного хода луча электронно-лучевой трубки. Это сводит к минимуму отклонения в отображении цветов, а также мерцание дисплея. Пока что в полной мере не поддерживается

По моему мнению, функция CreatePalette() имеет слишком много сбивающих с толку параметров. Мы будем работать в основном с 8-битовыми палитрами, поэтому для работы нам понадобятся такие флаги (объединенные побитовым оператором OR):

```
DDPCAPS_8BIT | DDPCAPS_ALLOW256 | DDPCAPS_INITIALIZE
```

Если записи 0 и 256 заполняются в соответствии с принятыми соглашениями, флаг DDPCAPS_ALLOW256 можно опустить. Более того, если в процессе вызова функции CreatePalette() не происходит передача палитры, можно забыть и о флаге DDPCAPS_INITIALIZE.

Поводя итог сказанному, приведем код, предназначенный для создания объекта-палитры на основе палитры, сформированной нами ранее из случайных цветов.

```
LPDIRECTDRAWPALETTE lpddpal = NULL; // Интерфейс палитры

if (FAILED(lpdd->CreatePalette(DDPCAPS_8BIT|
                  DDPCAPS_ALLOW256|
                  DDPCAPS_INITIALIZE,
                  palette,
                  &lpddpal,
                  NULL)))
{
  // ошибка
} // if
```

В случае успешного вызова функция с помощью параметра lpddpal возвращает интерфейс IDirectDrawPalette. Кроме того, аппаратная палитра цветов тут же обновляется переданной функции палитрой, которая в нашем случае представляет собой набор 256 случайных цветов.

В этом месте нужно было бы привести живой пример, однако, к сожалению, мы находимся на одном из переходных этапов изучения DirectDraw. Мы не сможем увидеть цвета, пока не научимся выводить их на экран. Поэтому перейдем к следующему теоретическому шагу.

Построение поверхности дисплея

Как известно, выводимое на экран монитора изображение — это не что иное, как матрица цветных пикселей, представленная в памяти в определенном формате: в формате палитры или RGB. В любом случае, чтобы что-нибудь произошло, нужно знать, как занести изображение в память. Однако разработчики DirectDraw решили немного абстрагировать концепцию видеопамяти, чтобы, с точки зрения программиста, доступ к *видеоповерхности* выглядел одинаково, какая бы причудливая видеокарта не была установлена в системе. Поэтому DirectDraw поддерживает структуры, которые называются *поверхностями* (surfaces). Как видно из рис. 6.7, поверхности — это прямоугольные области памяти, в которых могут храниться растровые данные. Кроме того, поверхности могут быть первичными или вторичными. Первичная поверхность непосредственно соответствует фактической видеопамяти, которая выводится видеокартой и содержание которой постоянно отображается на экране. Таким образом, в любой программе с использованием DirectDraw имеется только одна первичная поверхность; она направляется непосредственно на экран и обычно хранится в видеопамяти. Результаты преобразования первичной поверхности сразу же видны на экране. Например, установив видеорежим 640×480×256, необходимо создать первичную поверхность в этом же режиме, а затем присоединить ее к дисплею — объекту `IDirectDraw7`.

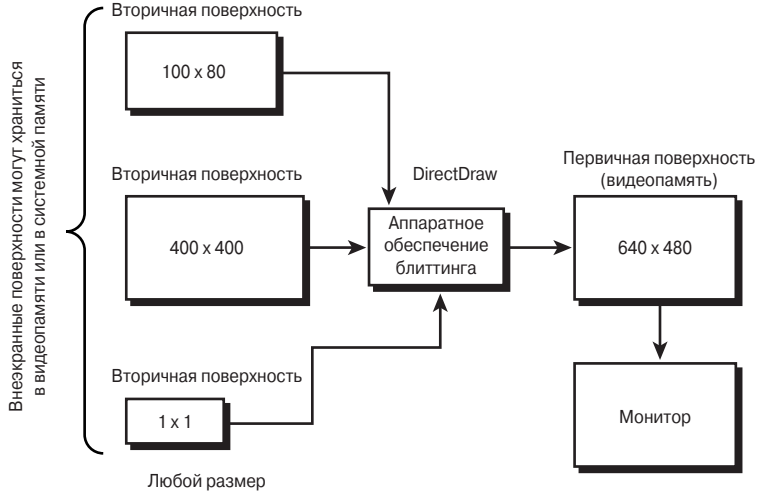


Рис. 6.7. Поверхности могут быть любых размеров

Вторичные поверхности намного более гибкие. Они могут иметь любой размер, находиться в видеопамяти или системной памяти, их можно создавать в любом количестве, какое только способна выдержать память. В большинстве случаев создается одна или две вторичные поверхности (задние буферы), для того чтобы обеспечить плавную анимацию. Глубина их цветов и геометрические размеры совпадают с соответствующими параметрами первичной поверхности. Затем эти внеэкранные поверхности обновляются следующим кадром анимации, после чего быстро копируются в первичную поверхность (*переключение страниц* (page flipping))

для обеспечения плавной анимации. Такой метод называется *двойной* или *тройной буферизацией* (double buffering или triple buffering). В следующей главе это рассматривается подробнее, а пока речь идет о применении вторичных поверхностей.

Другая область использования вторичных поверхностей — хранение в них битовых образов и анимации, которыми представлен объект в игре. Эта особенность технологии DirectDraw очень важна, потому что применять аппаратное ускорение при использовании растровых данных можно только с помощью этой технологии. Если программист сам пишет код для реализации операции *блиттинга* битов (передачи растрового изображения), теряются все возможности аппаратного ускорения.

А теперь будем двигаться дальше. Пора наконец узнать, как создаются первичные поверхности, которые имеют размер, совпадающий с разрешением дисплея. Кроме того, вы научитесь записывать данные в созданную первичную поверхность и выводить пиксели на экран.

Создание первичной поверхности

Чтобы создать первичную поверхность, выполните следующее.

1. Заполните структуру данных DDSURFACEDESC2, которая описывает создаваемую поверхность.
2. Для создания поверхности вызовите функцию IDirectDraw7::CreateSurface(), прототип которой приведен ниже.

```
HRESULT CreateSurface(
    LPDDSURFACEDESC2 lpDDSurfaceDesc2,
    LPDIRECTDRAWSURFACE4 FAR *lplpDDSurface,
    IUnknown FAR *pUnkOuter);
```

Эта функция получает описание создаваемой поверхности DirectDraw, указатель, предназначенный для хранения интерфейса и используемое нами для свойства модели COM pUnkOuter значение NULL. Процедура заполнения структуры данных может немного сбивать с толку, однако автор надеется, что его объяснения будут понятны. Начнем с определения DDSURFACEDESC2:

```
typedef struct _DDSURFACEDESC2
    {
    DWORD dwSize;  // Размер данной структуры
    DWORD dwFlags;  // Управляющие флаги
    DWORD dwHeight; // Высота поверхности в пикселях
    DWORD dwWidth;  // Ширина поверхности в пикселях
    union
    {
      LONG lPitch;                  // Шаг памяти, выделяемый для строки
      DWORD dwLinearSize;           // размер буфера в байтах
    } DUMMYUNIONNAMEN(1);
    DWORD dwBackBufferCount; // Количество задних буферов
    union
    {
      DWORD dwMipMapCount; // Количество уровней
      DWORD dwRefreshRate;   // Частота обновления
    } DUMMYUNIONNAMEN(2);
    DWORD dwAlphaBitDepth;   // Количество альфа-битов
    DWORD dwReserved;            // Зарезервировано
    LPVOID lpSurface;            // Указатель памяти поверхности
```

```
DDCOLORKEY ddckCKDestOverlay;
DDCOLORKEY ddckCKDestBlt;      // Целевой ключ цвета
DDPIXELFORMAT ddpfPixelFormat; // Формат пикселей
DDSCAPS2 ddsCaps;              // Характеристики поверхности
DWORD dwTextureStage;           // Для присоединения текстуры
                                // на определенной стадии работы с D3D
} DDSURFACEDESC2, FAR* LPDDSURFACEDESC2;
```

Как видите, структура довольно сложна, причем назначение трех четвертей ее полей — сплошная загадка. К счастью, вам нужно понимать значение только некоторых из них — тех, которые выделены жирным шрифтом. Рассмотрим их функции более подробно.

dwSize. Одно из важнейших полей в любой структуре DirectX. Многие структуры данных DirectX передаются в функцию по адресу, поэтому функция или метод, аргументом которых является структура данных, не имеет информации о ее размере. Однако если в первом 32-битовом значении всегда задавать размер структуры данных, то получающая ее функция всегда будет знать, сколько данных содержится в структуре: для этого достаточно посмотреть на первое значение DWORD. Таким образом, в качестве первого элемента всех структур DirectDraw и DirectX выступает спецификатор размера. Может показаться, что этот спецификатор не нужен, но на самом деле поместить его в структуры данных было хорошей идеей. Все, что нужно сделать, — это присвоить ему значение, например, так:

```
DDSURFACEDESC2 ddsd;
ddsd.Size = sizeof(DDSURFACEDESC2);
```

dwFlags. Это поле используется либо для того, чтобы указывать функциям DirectDraw, какие именно поля структуры будут заполняться достоверной информацией, либо (в операциях запроса) для указания того, какая именно информация запрашивается. В табл. 6.5 приведены битовые значения, которые могут использоваться во флаге. Например, если в поля dwWidth и dwHeight будут помещены достоверные значения, поле dwFlags можно задать так:

```
ddsd.dwFlags = DDSD_WIDTH| DDSD_HEIGTH;
```

После этого функция DirectDraw будет знать, что в полях dwWidth и dwHeight содержится достоверная информация. Поле dwFlags можно рассматривать как указатель на достоверные данные.

Таблица 6.5. Битовые значения поля dwFlags структуры DDSURFACEDESC2

Значение	*Описание*
DDSD_ALPHABITDEPTH	Указывает, что поле dwAlphaBitDepth содержит достоверное значение
DDSD_BACKBUFFERCOUNT	Указывает, что поле dwBackBufferCount содержит достоверное значение
DDSD_CAPS	Указывает, что поле ddsCaps содержит достоверное значение
DDSD_CKDESTBLT	Указывает, что поле ddckCKDestBlt содержит достоверное значение
DDSD_CKDESTOVERLAY	Указывает, что поле ddckCKDestOverlay содержит достоверное значение
DDSD_CKSRCBLT	Указывает, что поле ddckCKSrcBlt содержит достоверное значение
DDSD_CKSRCOVERLAY	Указывает, что поле ddckCKSrcOverlay содержит достоверное значение
DDSD_HEIGHT	Указывает, что поле dwHeight содержит достоверное значение
DDSD_LINEARSIZE	Указывает, что поле dwLinearSize содержит достоверное значение
DDSD_LPSURFACE	Указывает, что поле lpSurface содержит достоверное значение

Значение	*Описание*
DDSD_MIPMAPCOUNT	Указывает, что поле dwMipMapCount содержит достоверное значение
DDSD_PITCH	Указывает, что поле lPitch содержит достоверное значение
DDSD_PIXELFORMAT	Указывает, что поле ddpfPixelFormat содержит достоверное значение
DDSD_REFRESHRATE	Указывает, что поле dwRefreshRate содержит достоверное значение
DDSD_TEXTURESTAGE	Указывает, что поле dwTextureStage содержит достоверное значение
DDSD_WIDTH	Указывает, что поле dwWidth содержит достоверное значение

dwWidth. В этом поле указывается ширина поверхности в пикселях. Кроме того, при запросе свойств поверхности это поле возвращает ее ширину.

dwHeight. В этом поле указывается высота поверхности в пикселях. При запросе свойств поверхности это поле возвращает ее высоту.

lPitch. Очень интересное поле. Это размер блока памяти, который отводится под одну строку изображения при данном видеорежиме (рис. 6.8). Эту величину также называют *шагом* (stride) или *шириной памяти* (memory width). Как бы ни звучало название, суть в том, что значение этого поля — очень важная часть данных. Запрашивая определенный видеорежим, например 640×480×8, можно предположить, что в поле lPitch следует задать значение 640, так как в строке находится 640 пикселей а каждый пиксель занимает 8 бит (1 байт). Однако, как видно из рис. 6.8, эти соображения могут оказаться неправильными.

СОВЕТ

Осуществляя доступ к поверхности DirectDraw, важно помнить о том, что в силу особенностей архитектуры видеокарт значение lPitch может отличаться от произведения ширины видеорежима на количество байтов в пикселе. Поэтому при переходе от одной строки к другой следует пользоваться не этим произведением, а значением lPitch.

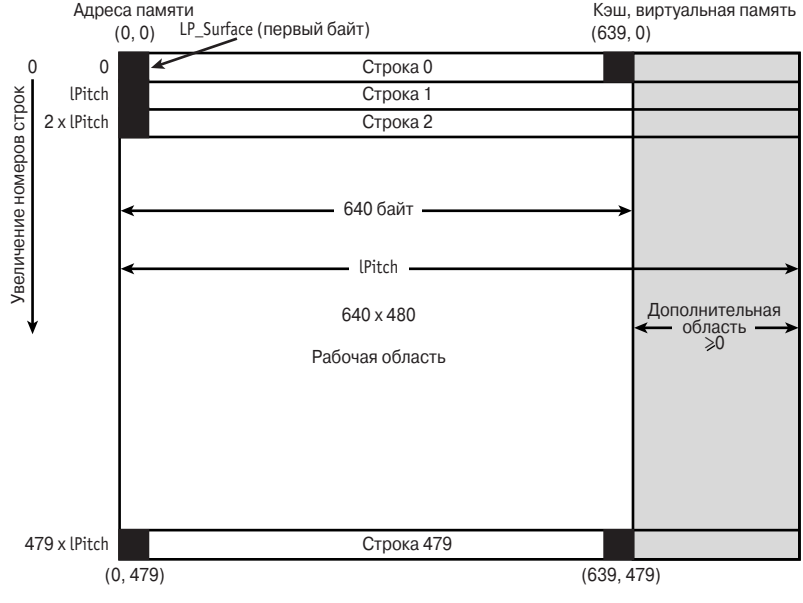


Рис. 6.8. Доступ к поверхности

Большинство новых видеокарт поддерживают так называемые *линейные режимы памяти* (linear memory modes), а также имеют специальные устройства адресации. Для таких видеокарт шаг памяти равен произведению количества пикселей в строке на количество байтов в пикселе, но считать, что это равенство гарантированно соблюдается, нельзя. Поэтому предположение о том, что в видеорежиме 640×480×8 длина строки равна 640 байт, может оказаться неверным. Вот зачем нужно поле lPitch. Чтобы вычисления, связанные с адресацией памяти, оказались правильными и переход от одной строки к другой был выполнен корректно, необходимо обращаться к этому полю. Например, чтобы осуществить доступ к определенному пикселю в видеорежиме 640×480×8 (256 цветов), можно воспользоваться следующим кодом:

```
ddsd.lpSurface[x + y*ddsd.lPitch] = color;
```

При этом предполагается, что к данному моменту были вызваны функции DirectDraw, возвращающие значения lPitch и lpSurface (это поле указывает на занимаемую поверхностью область памяти; подробнее оно описывается ниже). Вернемся к приведенной выше строке. Она довольно проста, не так ли? В большинстве случаев в режиме 640×480×8 значение ddsd.lPitch равно 640, а в режиме 640×480×16 — 1280 (по два байта на пиксель, 640×2 = 1280). Однако для некоторых видеокарт это утверждение может оказаться неверным из-за особенностей использования памяти этими картами, наличия в них внутреннего кэша или по каким-то иным причинам. Итак, в расчетах, связанных с памятью, всегда используйте значение lPitch, и тогда ваш код будет надежно работать.

СЕКРЕТ

Несмотря на то что значение lPitch иногда отличается от горизонтальной величины разрешения, может иметь смысл запросить его для того, чтобы использовать функции с более оптимизированным кодом. Например, значение lPitch можно получить в процессе инициализации кода программы, а затем сравнить его с горизонтальной величиной разрешения. Если они равны, то появляется возможность воспользоваться высокооптимизированным кодом с жестко запрограммированным количеством байтов в строке.

lpSurface. Это поле применяется для запроса указателя на физическую область памяти, в которой хранится создаваемая поверхность. Эта область может находиться в видеопамяти или в системной памяти, однако конкретное ее расположение не так важно. Имея в распоряжении указатель на эту область, программист получает возможность выполнять в ней те же операции, что и в любой другой области памяти — записывать туда информацию, считывать ее и т.д. Именно по такому принципу и осуществляется вывод пикселей на экран. К сожалению, чтобы получить достоверное значение этого указателя, необходимо проделать определенную работу, к описанию которой мы перейдем несколько позже. По сути, необходимо "заблокировать" область памяти, в которой хранится поверхность, и сообщить функциям DirectX, что вы собираетесь с ней работать и что никакой другой процесс не должен выполнять операции чтения из этой памяти или записи в нее. Кроме того, получив этот указатель, его, как правило, требуется привести к определенному типу (зависящему от глубины цвета) и присвоить это значение рабочему указателю.

dwBackBufferCount. Данное поле применяется, чтобы установить или считать количество задних буферов или вторичных внеэкранных буферов, выстроенных в виде последовательности "позади" первичной поверхности. Напомним, что задние буферы используются для реализации плавной анимации путем создания одного или нескольких виртуальных первичных буферов (с одинаковыми геометрическими размерами и глубиной цветов), которые находятся вне экрана. Затем мы берем этот невидимый пользователю задний буфер и выполняем быстрое копирование его в первичный буфер, содержимое

которого выводится непосредственно на экран. Если создается только один задний буфер, то такая технология называется *двойной буферизацией* (double buffering). Метод, при котором применяется два задних буфера, называется *тройной буферизацией* (triple buffering); он работает несколько быстрее, но интенсивнее расходует память. Для простоты мы обычно будем создавать обращаемые последовательности из одной первичной поверхности и одного заднего буфера.

ddckCKDestBlt. Это поле применяется для управления целевым ключом цвета, используемого в операциях блиттинга. Подробнее это описывается в главе 7, "Постигаем секреты DirectDraw и растровой графики".

ddckCKScrBlt. Это поле применяется для указания исходного ключа цвета. По сути, это те цвета, которые не нужно блиттировать в процессе формирования битового образа. Таким образом задаются прозрачные цвета. Подробнее это рассматривается в главе 7, "Постигаем секреты DirectDraw и растровой графики".

ddptfPixelFormat. Это поле применяется для получения формата пикселей, из которых состоит поверхность, что важно для понимания ее свойств. Ниже приводится общий вид этой структуры. Чтобы узнать о ней подробнее, обратитесь к документации DirectX SDK, так как ее описание довольно объемно и приводить его здесь неуместно.

```
typedef struct _DDPIXELFORMAT
{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwFourCC;
  union
  {
  DWORD dwRGBBitCount;
  DWORD dwYUVBitCount;
  DWORD dwZBufferBitDepth;
  DWORD dwAlphaBitDepth;
  DWORD dwLuminanceBitCount;  // Новый член DirectX 6.0
  DWORD dwBumpBitCount;       // Новый член DirectX 6.0
  } DUMMYUNIONNAMEN(1);
  union
  {
  DWORD dwRBitMask;
  DWORD dwYBitMask;
  DWORD dwStencilBitDepth;    // Новый член DirectX 6.0
  DWORD dwLuminanceBitMask;   // Новый член DirectX 6.0
  DWORD dwBumpDuBitMask;      // Новый член DirectX 6.0
  } DUMMYUNIONNAMEN(2);
  union
  {
  DWORD dwGBitMask;
  DWORD dwUBitMask;
  DWORD dwZBitMask;           // Новый член DirectX 6.0
  DWORD dwBumpDvBitMask;      // Новый член DirectX 6.0
  } DUMMYUNIONNAMEN(3);
  union
  {
  DWORD dwBBitMask;
  DWORD dwVBitMask;
  DWORD dwStencilBitDepth;    // Новый член DirectX 6.0
```

```
  DWORD dwBumpLuminanceBitMask;// Новый член DirectX 6.0
  } DUMMYUNIONNAMEN(4);
  union
  {
  DWORD dwRGBAlphaBitMask;
  DWORD dwYUVAlphaBitMask;
  DWORD dwLuminanceAlphaBitMask; // Новый член DirectX 6.0
  DWORD dwRGBZBitMask;
  DWORD dwYUVZBitMask;
  } DUMMYUNIONNAMEN(5);
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;
```

НА ЗАМЕТКУ Некоторые наиболее часто используемые поля выделены жирным шрифтом.

ddsCaps. Это поле применяется для указания запрашиваемых свойств поверхности, созданной в каком-то другом месте программы. Оно представляет собой сложную структуру данных DDSCAPS2.

```
typedef struct _DDSCAPS2
{
  DWORD dwCaps;  // Особенности поверхности
  DWORD dwCaps2; // Дополнительные особенности поверхности
  DWORD dwCaps3; // Объяснения будут даны позже
  DWORD dwCaps4; // Объяснения будут даны позже
} DDSCAPS2, FAR* LPDDSCAPS2;
```

В 99.9% случаев задается только первое поле dwCaps. Поле dwCaps2 предназначено для программирования трехмерной графики. Объяснение смысла полей dwCaps3 и dwCaps4 приведено ниже, а сами они в этой книге не используются. В табл. 6.6 содержится неполный список возможных значений флага dwCaps (полный список можно найти в документации DirectX SDK).

Например, при создании первичной поверхности значение переменной-члена ddsd.ddsCaps можно задать так:

```
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

Все это может показаться слишком сложным, и в какой-то мере так это и есть. Наличие управляющих флагов с двойным вложением несколько огорчает, но с этим ничего не поделаешь.

Таблица 6.6. Флаги свойств поверхностей DirectDraw

Значение	*Описание*
DDSCAPS_BACKBUFFER	Указывает, что данная поверхность — задний буфер структуры, состоящей из обращаемых поверхностей
DDSCAPS_COMPLEX	Указывает, что описывается сложная поверхность, т.е. поверхность, состоящая из одной первичной поверхности и одного или нескольких задних буферов; с помощью сложной поверхности создается последовательная цепочка изображений
DDSCAPS_FLIP	Указывает, что данная поверхность входит в состав структуры, состоящей из сменяющих друг друга поверхностей. При передаче этого свойства функции-методу CreateSurface() создаются один передний и один или несколько задних буферов

Окончание табл. 6.6

Значение	*Описание*
DDSCAPS_LOCALVIDMEM	Указывает, что данная поверхность хранится в локальной видеопамяти. Если этот флаг задан, должен быть также установлен флаг DDSCAPS_VIDEOMEMORY
DDSCAPS_MODE X	Указывает, что данная поверхность создана в режиме 320×200 или 320×240 Mode X
DDSCAPS_NONLOCALVIDMEM	Указывает, что данная поверхность хранится в нелокальной видеопамяти. Если этот флаг установлен, должен также быть установлен флаг DDSCAPS_VIDEOMEMORY
DDSCAPS_OFFSCREENPLAIN	Указывает, что данная поверхность является внеэкранной, отличной от таких особых поверхностей, как наложение (overlay), текстура, z-буфер, передний буфер, задний буфер или альфа-поверхность. Этот флаг обычно используется для спрайтов
DDSCAPS_OWNDC	Указывает, что данная поверхность в течение длительного времени будет содержать контекст устройства
DDSCAPS_PRIMARYSURFACE	Указывает, что данная поверхность первичная. Этот флаг представляет то, что пользователь видит в данный момент
DDSCAPS_STANDARDVGAMODE	Указывает, что данная поверхность задана в стандартном режиме VGA, а не в режиме Mode X. Этот флаг нельзя использовать совместно с флагом DDSCAPS_MODE X
DDSCAPS_SYSTEMMEMORY	Указывает, что область памяти, отведенная для хранения данной поверхности, находится в системной памяти
DDSCAPS_VIDEOMEMORY	Указывает, что данная поверхность хранится в видеопамяти

Теперь, получив представление о возможностях, предоставляемых технологией DirectDraw при создании поверхности, а также о возникающих при этом сложностях, применим наши знания на практике и создадим простую первичную поверхность, размер и глубина цветов которой совпадают с соответствующими параметрами режима визуального отображения (поведение по умолчанию). Ниже приведен код, предназначенный для создания первичной поверхности.

```
// Указатель, предназначенный для хранения первичной
// поверхности. Заметим, что это 7-я версия интерфейса
LPDIRECTDRAWSURFACE7 lpddsprimary = NULL;

DDSURFACEDESC2 ddsd; // Описание поверхности DirectDraw

// Microsoft рекомендует очищать эту структуру
memset(&ddsd,0,sizeof(ddsd));
// (можно также использовать функцию ZeroMemory())

// Теперь зададим размер структуры
ddsd.dwSize = sizeof(ddsd);

// В нашем случае используется только поле ddsCaps.
// Можно было бы использовать поля, отвечающие за ширину,
// высоту и другие параметры поверхности, но они не нужны,
// поскольку размеры создаваемой первичной поверхности
```

```
// совпадают с размерами по умолчанию
ddsd.dwFlags = DDSD_CAPS;

// Задаем свойства из табл. 6.6
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Создаем первичную поверхность
if (FAILED(lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL)))
{
  // ошибка
} // if
```

В случае успешного вызова функции CreateSurface() в переменной lpddsprimary будет содержаться указатель на интерфейс новой поверхности, с помощью которого можно будет вызывать различные методы (их не так много; в качестве примера можно привести присоединение палитры в режимах с 256 цветами). Рассмотрим это действие подробнее, чтобы довести до логического завершения приведенный выше пример по созданию палитры.

Присоединение палитры

В одном из предыдущих разделов, в котором шла речь о создании палитр, было описано почти все, за исключением присоединения палитры к поверхности. Палитра была создана, и все ее записи заполнены, однако присоединить эту палитру к поверхности не было возможности, поскольку самой поверхности тогда еще не было. Теперь, когда у нас есть поверхность (первичная), можно выполнить этап присоединения.

Для присоединения палитры к любой поверхности необходимо воспользоваться приведенной ниже функцией IDirectDrawSurface7::SetPalette().

```
HRESULT SetPalette(LPDIRECTDRAWPALETTE lpDDPalette);
```

Эта функция получает указатель на присоединяемую палитру. Чтобы присоединить созданную ранее палитру к нашей первичной поверхности, воспользуемся следующим кодом:

```
if (FAILED(lpddsprimary->SetPalette(lpddpal)))
{
  // ошибка
} // if
```

Теперь у вас есть все, что может потребоваться для эмуляции полноценных возможностей, которыми обладают игры, написанные под DOS32. Вы научились переключать видеорежимы, настраивать палитры и создавать поверхности, представляющие активные видеоизображения. Однако это еще не все: вам предстоит узнать практический рецепт блокирования области памяти, в которой хранится первичная поверхность, получения доступа к видеопамяти и вывода пикселей на экран. Сейчас мы этим и займемся.

Вывод пикселей на экран

Чтобы вывести пиксель (или несколько пикселей) в полноэкранном режиме работы DirectDraw, сначала необходимо настроить DirectDraw, установить уровень совместного доступа, установить видеорежим и создать, как минимум, первичную поверхность. После этого нужно получить доступ к первичной поверхности и внести записи в видеопамять. Однако, прежде чем вы научитесь это делать, рассмотрим принципы работы видеоповерхностей с другой точки зрения.

Напомним, что видеорежимы и поверхности являются линейными (рис. 6.9). Это означает, что номера ячеек памяти возрастают слева направо, а при переходе от одной строки к другой — сверху вниз.

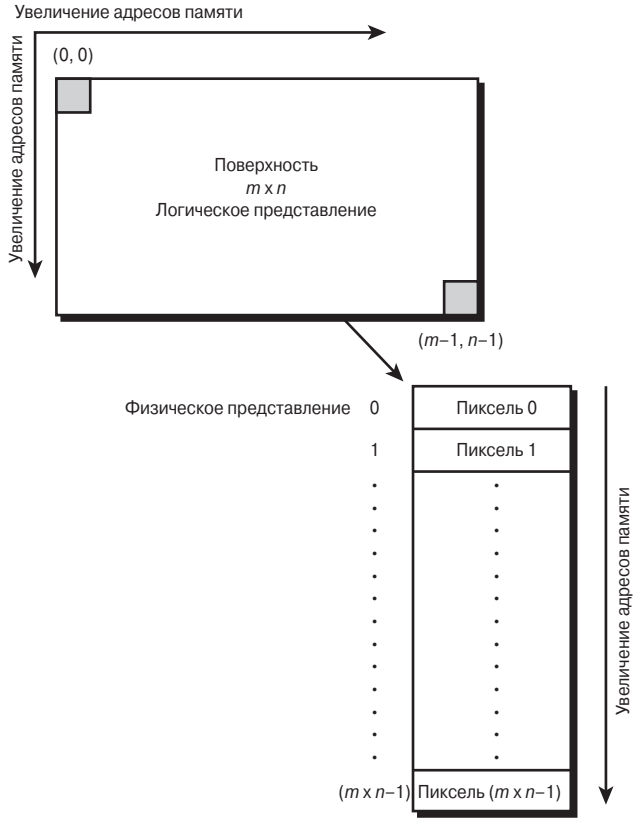


Рис. 6.9. Поверхности DirectDraw линейны

СОВЕТ

Вы можете спросить, как это в технологии DirectDraw нелинейный видеорежим чудесным образом преобразуется в линейный, хотя самой видеокартой такой переход не поддерживается. Например, режим Mode X полностью нелинеен и работает с использованием коммутации блоков памяти. На самом деле в ситуациях, когда DirectDraw обнаруживает, что в аппаратном устройстве данный режим является нелинейным, вызывается драйвер VFLATD.VXD, создающий между программистом и видеопамятью программный промежуточный уровень и позволяющий считать видеопамять линейной. Однако следует иметь в виду, что использование этого драйвера замедляет работу программы.

Чтобы указать любую позицию в буфере изображения, достаточно задать два параметра: шаг памяти, который отводится для хранения одной строки (т.е. сколько байтов занимает каждая строка), а также размер каждого пикселя (8, 16, 24 или 32 бит). Можно воспользоваться следующей формулой:

```
// Считаем, что приведенный ниже указатель указывает на
// видеопамять или область памяти, отведенную поверхности
UCHAR *video_buffer8;

video_buffer8[x + y*memory_pitchB] = pixel_color_8;
```

Конечно же, приведенная формула не совсем верна, так как она работает только для режимов, в которых пиксель занимает один байт. Для тех видеорежимов, в которых пиксель занимает два байта, нужно воспользоваться следующим кодом:

```
// Считаем, что приведенный ниже указатель указывает на
// видеопамять или область памяти, отведенную поверхности
USHORT *video_buffer16;

video_buffer16[x + y*(memory_pitchB >> 1)] = pixel_color_16;
```

В приведенных выше строках выполняется много действий, поэтому рассмотрим их подробнее. Поскольку задан 16-битовый режим, в коде используется указатель на видеопамять USHORT. Это позволяет получать доступ к массиву, однако при этом следует применять 16-битовую арифметику. Таким образом, инструкция

```
video_buffer16[1]
```

на самом деле предоставляет доступ ко второй группе байтов типа SHORT, или ко второму и третьему байтам. Кроме того, поскольку значение переменной memory_pitchB выражено в байтах, его необходимо поделить пополам. Это достигается путем сдвига на один бит вправо. Наконец, оператор присвоения значения pixel_color_16 тоже может ввести в заблуждение, так как теперь в видеопамять заносится полное 16-битовое значение типа USHORT, а не 8-битовое значение, как в предыдущем случае. Более того, 8-битовое значение может быть индексом цвета, в то время как 16-битовое должно быть значением RGB, закодированным, как правило, в формате $R_5G_6B_5$. Эта запись означает, что для красного цвета отводится пять битов, для зеленого — шесть, для синего — пять (рис. 6.10).

Далее приведен текст макросов, предназначенных для создания 16-битового слова RGB в формате 5.5.5 и в формате 5.6.5.

```
// Создание 16-битового значения цвета
// в формате 5.5.5 (1-битовый альфа-режим)
#define _RGB16BIT555(r,g,b) \
   ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))

// Создание 16-битового значения цвета
// в формате 5.6.5 (режим с приоритетом зеленого)
#define _RGB16BIT565(r,g,b) \
   ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))
```

Рис. 6.10. Возможные 16-битовые кодировки RGB, включая формат 5.6.5

Как видите, в целом адресация и принципы преобразования для 16-битовых режимов и режимов RGB ненамного сложнее, чем для 8-битовых режимов с 256 цветами.

Теперь можно приступать к выводу пикселей на экран.

Чтобы получить доступ к любой поверхности — первичной, вторичной или какой-нибудь другой, необходимо заблокировать (а потом разблокировать) область памяти, отведенную для этой поверхности. Такая последовательность блокирования и разблокирования необходима по двум причинам: во-первых, чтобы сообщить DirectDraw, что вы собираетесь работать с этой памятью (т.е. другим процессам не следует предоставлять к ней доступ); во-вторых, чтобы указать видеоустройствам, что они не должны перемещать никаких буферов кэша или виртуальной памяти, пока ведется работа с заблокированной памятью. Следует помнить, что видеопамять, вообще говоря, не остается на одном и том же месте — она вполне может быть виртуальной. Если же эту память заблокировать, она гарантированно остается в одном и том же адресном пространстве в течение времени блокировки. Функция, предназначенная для блокировки памяти, называется IDirectDrawSurface7::Lock(). Ниже приведен ее прототип.

```
HRESULT Lock(
  LPRECT lpDestRect, // Блокируемая область
  LPDDSURFACEDESC2 lpDDSurfaceDesc,
                     // адрес структуры,
                     // получающей информацию
  DWORD dwFlags,     // флаги запроса
  HANDLE hEvent);    // У нас - всегда NULL
```

Рассмотрим параметры функции подробнее. Первый параметр — это запираемая прямоугольная подобласть той области памяти, которая выделена для поверхности (рис. 6.11). Технология DirectDraw позволяет заблокировать не всю выделенную для поверхности область, а только ее часть, чтобы могли продолжать выполняться процессы, которым нужен доступ только к незаблокированной области памяти. Это очень полезно в тех случаях, когда программист намерен обновить лишь определенную часть поверхности и ему не нужно блокировать ее полностью. Однако в большинстве случаев для простоты блокируется вся поверхность — для этого нужно всего лишь передать в первом параметре значение NULL.

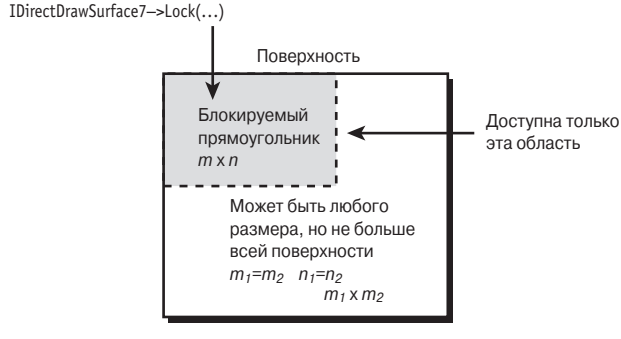


Рис. 6.11. Действие функции IDirectDrawSurface7->Lock(...)

Второй параметр — это адрес структуры DDSURFACEDESC2, которую нужно заполнить информацией о запрашиваемой поверхности. По сути, в этот параметр достаточно передать пустую структуру DDSURFACEDESC2. Следующий параметр, dwFlags, сообщает функции Lock(), что именно она должна делать. В табл. 6.7 приведен список наиболее часто используемых значений этого флага.

Таблица 6.7. Управляющие флаги метода Lock()

Значение	*Описание*
DDLOCK_READONLY	Указывает, что запираемая область будет доступна только для чтения
DDLOCK_SURFACEMEMORYPTR	Указывает, что функция должна возвратить корректный указатель на вершину заданного прямоугольника. Если прямоугольник не задан, по умолчанию возвращается указатель на вершину поверхности
DDLOCK_WAIT	Если блокировку выполнить невозможно из-за активности операции пересылки видеопамяти, осуществляются повторные попытки до тех пор, пока не удастся заблокировать память или пока не произойдет другая ошибка, например DDERR_SURFACEBUSY
DDLOCK_WRITEONLY	Указывает, что запираемая область будет доступна для записи

Примечание. Некоторые чаще всего используемые поля выделены жирным шрифтом.

Последний параметр используется для реализации дополнительных возможностей, а именно событий Win32. Мы всегда используем здесь значение NULL.

Заблокировать первичную поверхность очень просто: нужно лишь запросить указатель на эту поверхность и попросить DirectDraw подождать, пока она станет доступна. Ниже приведен соответствующий код.

```
DDSURFACEDESC2 ddsd;
// Здесь будут храниться результаты блокировки

// Очистка данных, описывающих поверхность
memset(&ddsd, 0, sizeof(ddsd));

// Установка поля размера
ddsd.dwSize = sizeof(ddsd);

// Блокировка поверхности
if (FAILED(lpddsprimary->Lock(NULL,
       &ddsd,
       DDLOCK_SURFACEMEMORYPRT| DDLOCK_WAIT, NULL)))
{
  // ошибка
} // if

// ****** Сейчас нас интересуют два поля: ddsd.lPitch,
// содержащее шаг памяти (в байтах), и ddsd.lpSurface,
// являющееся указателем на верхний левый угол
// блокируемой поверхности
```

Заблокировав поверхность, программист получает возможность работать с отведенной для нее областью памяти по своему усмотрению. Шаг памяти, который отводится для строки, хранится в переменной ddsd.lPitch, а указатель на саму поверхность — в переменной ddsd.lpSurface. Поэтому в 8-байтовом режиме (1 байт на пиксель) для вывода пикселя в любое место первичной поверхности можно использовать такую функцию:

```
inline void Plot8(
  int x, int y,  // Координаты пикселя
```

```
  UCHAR  color,    // Индекс цвета пикселя
  UCHAR *buffer,   // Указатель на память поверхности
  int    mempitch) // Шаг памяти, отведенный для линии
{
  // эта функция выводит один пиксель
  buffer[x+y*mempitch] = color;
} // Plot8
```

Вызов этой функции, предназначенный для вывода пикселя с координатами (100, 20) и индексом цвета 26, выглядит так:

```
Plot8(100,20,26, (UCHAR *)ddsd.lpSurface,(int)ddsd.lPitch);
```

Аналогично можно написать функцию для вывода пикселя в режиме RGB 5.6.5:

```
inline void Plot16(
  int x, int y,         // Координаты пикселя
  UCHAR red,
  UCHAR green,
  UCHAR, blue       // Цвет пикселя в формате RGB
  USHORT *buffer,  // Указатель на память поверхности
  int mempitch)    // Шаг памяти строки
{
  // функция выводит один пиксель
  buffer[x+y*(mempitch>>1)] = _RGB16BIT565(red,green,blue);
} // конец функции Plot16
```

А вот как можно вывести пиксель с координатами (300, 100) и значениями RGB, равными (10, 14, 30):

```
Plot16(300,100,10,14,30,
    (USHORT *)ddsd.lpSurface,
    (int)ddsd.lPitch);
```

Завершив работу по созданию очередного кадра анимации, поверхность нужно разблокировать. Это делается с помощью приведенного ниже метода IDirectDrawSurface7::Unlock.

```
HRESULT Unlock(LPRECT lpRect);
```

В функцию Unlock() передается исходное значение RECT, которое было использовано в команде блокирования, или значение NULL, если блокировалась вся поверхность. В последнем случае для разблокирования поверхности достаточно поместить в программу такой код:

```
if (FAILED(lpddsprimary->Unlock(NULL)))
{
  // ошибка
} // if
```

Теперь у нас есть все, что нужно для вывода изображений. Рассмотрим все шаги, необходимые для вывода на экран случайного пикселя (код, предназначенный для выявления ошибок, опущен).

```
LPDIRECTDRAW7 lpdd = NULL;
      // Интерфейс DirectDraw 7.0
LPDIRECTDRAWSURFACE7 lpddsprimary = NULL;
      // указатель на поверхность
DDSURFACEDESC2 ddsd;
      // Описание поверхности
```

```
LPDIRECTDRAWPALETTE lpddpal = NULL;
        // Интерфейс палитры
PALETTEENTRY palette[256];
        // Хранилище палитры

// Создание базового интерфейса IDirectDraw 7.0
DirectDrawCreateEx(NULL, (void **)&lpdd,
              IID_IDirectDraw7, NULL);

// Установка взаимодействия для полноэкранного режима
lpdd7->SetCooperativeLevel(hwnd,
                    DDSCL_FULLSCREEN|
                    DDSCL_ALLOWMODEX|
                    DDSCL_EXCLUSIVE|
                    DDSCL_ALLOWREBOOT);

// Установка режима монитора 640x480x256
lpdd->SetDisplayMode(640,480,8,0,0);

// Обнуление переменной ddsd и установка размера
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);

// Задание полей
ddsd.dwFlags = DDSD_CAPS;

// Запрос первичной поверхности
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Создание первичной поверхности
lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL);


// Создание массива для хранения данных палитры
for(int color=1; color < 255; color++)
{
   // Заполнение случайными значениями RGB
   palette[color].peRed   = rand()%256;
   palette[color].peGreen = rand()%256;
   palette[color].peBlue  = rand()%256;

   // Установка флага PC_NOCOLLAPSE
   palette[color].peFlags = PC_NOCOLLAPSE;
} // for color

// заполнение элементов 0 и 255 черным и белым цветами
palette[0].peRed = 0;
palette[0].peGreen = 0;
palette[0].peBlue = 0;
palette[0].peFlags = PC_NOCOLLAPSE;

palette[255].peRed = 255;
palette[255].peGreen = 255;
palette[255].peBlue = 255;
palette[255].peFlags = PC_NOCOLLAPSE;
```

```
// Создание объекта-палитры
lpdd->CreatePalette(DDPCAPS_8BIT | DDPCAPS_ALLOW256 |
            DDPCAPS_INITIALIZE,
            palette,&lpddpal, NULL)

// Присоединение палитры к первичной поверхности
lpddsprimary->SetPalette(lpddpal);

// Блокируем поверхность и запрашиваем указатель и шаг памяти

// Обнуление переменной ddsd и установка размера
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);

lpddsprimary->Lock(NULL, &ddsd,
        DDLOCK_SURFACEMEMORYPRT| DDLOCK_WAIT, NULL);

// Теперь поля ddsd.lPitch и ddsd.lpSurface достоверны

// Введение некоторых дополнительных переменных, чтобы код был
// более понятным и не требовалось преобразование типов
int mempitch      = ddsd.lPitch;
UCHAR *video_buffer = ddsd.lpSurface;

// Вывод на первичную поверхность 1000 пикселей со случайными
// координатами и цветами; они тут же отображаются на экране
for (int index=0; index < 1000; index++)
{
  // Выбор случайного положения и цвета для режима 640x480x8
  UCHAR color = rand()%256;
  int x = rand()%640;
  int y = rand()%480;

  // Вывод пикселя
  video_buffer[x+y*mempitch] = color;

} // for index

// Снятие блокировки с первичной поверхности
lpddsprimary->Unlock(NULL);
```

Конечно же, в приведенном выше коде опущена инициализация Windows и элементы, связанные с циклом событий, однако эти фрагменты программы всегда остаются неизменными. Для полноты картины рассмотрите демонстрационную программу DEMO6_3.CPP на прилагаемом компакт-диске. В ней содержится приведенный выше код, помещенный в функцию Game_Main(). На рис. 6.12 показан вид экрана при запуске программы.

При работе с программой обратите внимание на то, что в ней вместо стиля окна WS_OVERLAPPEDWINDOW использован стиль WS_POPUP. Напомним, что этот стиль лишен всех элементов управления и графического интерфейса пользователя Windows, что и требуется для полноэкранного приложения DirectX.

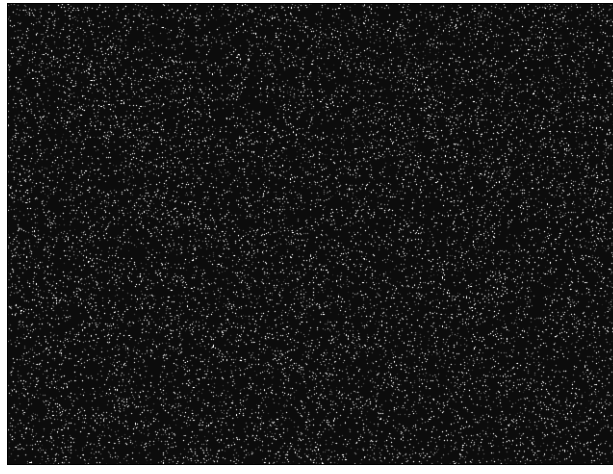


Рис. 6.12. Программа DEMO6_3.EXE в действии

Очистка

Прежде чем закончить главу, обратимся к теме, которая до сих пор откладывалась; речь идет об управлении ресурсами. Здесь, кстати, нет ничего сложного — просто после использования каждого объекта DirectDraw и DirectX к ним нужно применить функцию Release(). Например, в приведенном ниже исходном коде из функции Game_Shutdown() программы DEMO6_3.CPP можно увидеть несколько вызовов этой функции, предназначенных для того, чтобы избавиться от всех объектов DirectDraw и передать их обратно операционной системе и самой модели DirectDraw.

```
int Game_Shutdown(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз при завершении работы
// игры. В ней выполняются все действия по освобождению
// захваченных ресурсов и т.п.

// Сначала палитра
if (lpddpal)
{
   lpddpal->Release();
   lpddpal = NULL;
} // if

// Затем первичная поверхность
if (lpddsprimary)
{
   lpddsprimary->Release();
   lpddsprimary = NULL;
} // if

// Теперь избавляемся от интерфейса IDirectDraw7
if (lpdd)
{
   lpdd->Release();
```

```
    lpdd = NULL;
} // if

// Успешное завершение (или ваш собственный код)
return(1);

} // Game_Shutdown
```

Общее правило состоит в том, что после работы с объектами от них следует избавляться, причем делать это нужно в порядке, обратном порядку их создания. Например, если созданы объект DirectDraw, первичная поверхность и палитра (именно в таком порядке), то избавляться от них следует в обратном порядке: палитра, поверхность, а затем объект DirectDraw, как и было сделано в приведенном выше фрагменте кода.

ВНИМАНИЕ

Перед вызовом функции Release() убедитесь, что значение указателя интерфейса отлично от NULL. Это совершенно необходимо, так как удаление с использованием нулевого указателя легко приводит к проблемам.

Резюме

В этой главе вы ознакомились с основами технологии DirectDraw: научились выполнять подготовительные действия и запускать приложения в полноэкранном режиме. Кроме того, получены начальные представления о работе с палитрами, поверхностями экрана, а также о различиях между полноэкранными и оконными приложениями. В следующей главе интенсивность изучения материала будет гораздо выше и вы сможете получить еще больше важных и нужных знаний!

ГЛАВА 7

Постигаем секреты DirectDraw и растровой графики

В этой главе вы ознакомитесь с внутренним устройством технологии DirectDraw, после чего сможете приступить к работе с первым модулем графической библиотеки (T3DLIB1.CPP|H), который послужит основой всех демонстрационных примеров и игр, представленных в этой книге. Здесь освещено большое количество материала по DirectDraw и кратко описана графическая библиотека, созданная мною в процессе написания главы. И хотя материал главы довольно прост, его все же достаточно для того, чтобы, ознакомившись с ним, можно было сразу же делать что-то интересное. Ниже приведен список тем, раскрытых в этой главе.

- Высокоцветные режимы
- Переключение страниц и двойная буферизация
- Блиттер
- Отсечение графического изображения
- Загрузка битовых образов
- Анимация цветов
- Оконный режим приложений DirectX
- Получение информации от DirectX

Высокоцветные режимы

Безусловно, высокоцветные режимы (т.е. такие, в которых на каждый пиксель приходится больше 8 бит) имеют более привлекательный внешний вид, чем режимы с 256 цветами. Однако по ряду причин они обычно не применяются в программных реализациях трехмерной графики. Далее приведены самые важные из этих причин.

- **Скорость вычислений.** Стандартный буфер кадров в режиме 640×480 состоит из 307200 пикселей. Каждый пиксель состоит из 8 бит, поэтому большинство вычислений можно выполнить, выделяя по одному байту на пиксель, что упрощает растеризацию. В 16- и 24-битовом режимах вычисления обычно проводятся в полном RGB-пространстве (или с привлечением очень больших таблиц преобразования), и скорость этих вычислений снижается, как минимум, в два раза. Кроме того, в этих режимах в буфер кадров нужно записывать два или три пикселя вместо одного в 8-битовых режимах.

  Конечно же, при наличии аппаратных ускоряющих устройств это не такая уж большая проблема, если речь идет о формировании битового образа или трехмерного изображения (фактически большинство 3D-карт работают с 24- или 32-битовыми цветами), однако программная растеризация (выполнять которую вы как раз и учитесь в этой книге) обернется большими затратами ресурсов. Чтобы получить производительную программу, нужно стараться записывать как можно меньше данных, приходящихся на пиксель. 8-битовый режим удовлетворяет таким требованиям, хотя и выглядит не так привлекательно, как 16-битовый. Однако, создавая игру в 8-битовом режиме, можно быть уверенным, что в нее смогут играть и на компьютерах типа Pentium 133-233, и круг ваших пользователей не будет ограничен счастливыми владельцами компьютеров с процессором Pentium IV 2.4 GHz и 3D-ускорением.

- **Ограниченная скорость обмена с памятью.** Этот фактор редко принимается во внимание. В персональных компьютерах встречаются системные шины типа ISA (Industry Standard Architecture), VLB (VESA Local Bus), PCI (Peripheral Component Interconnect) или гибрид PCI/AGP (Accelerated Graphics Port). Все перечисленные порты, кроме AGP, работают по сравнению с тактовым генератором видеосистемы очень медленно. Это означает, что компьютер Pentium III 500+ MHz не сможет быстро работать с графикой, если на нем установлен порт PCI, ограничивающий скорость доступа к видеопамяти и/или ускоряющим устройствам. Конечно же, существуют различные способы оптимизации аппаратной конфигурации, такие как кэширование, видеопамять с несколькими портами и т.д., но при этом всегда остается ограничение по скорости заполнения, которое не удается преодолеть никакими ухищрениями. Отсюда вывод: при переходе к все более высоким разрешению и насыщенности цветов во многих случаях замедление работы программы больше связано с низкой скоростью памяти, чем со скоростью процессора. Однако при наличии порта AGP 2× или 4× актуальность этого утверждения снижается.

Тем не менее современные компьютеры являются достаточно мощными, чтобы работать с 16- и даже 24-битовыми программными средствами (которые достаточно быстры, хотя, конечно, не настолько, как аппаратные средства). Итак, создавая несложную игру, предназначенную для широкой аудитории, стоит подумать об использовании в ней 8-битового режима, который, помимо прочих достоинств, концептуально более прост для начинающих программистов. Принципы работы с высокоцветными режимами похожи на принципы работы с режимами с палитрой. Единственное, о чем не следует забывать, — это о том, что в кадровый буфер заносятся не индексы цветов, а полные значения пикселей в кодировке RGB. Это означает, что для тех режимов, с которыми предстоит работать, необходимо уметь создавать кодировку пикселей в формате RGB. Несколько 16-битовых кодировок RGB приведены на рис. 7.1.

Шестнадцатибитовый высокоцветный режим

Из рис. 7.1 видно, что имеется несколько возможных кодировок 16-битовых режимов, таких, как Alpha.5.5.5, X.5.5.5 и 5.6.5. Опишем их подробнее.

Alpha.5.5.5. В этом режиме один бит в позиции D_{15} представляет возможный альфа-компонент (прозрачность), а остальные 15 битов равномерно распределены между красным, зеленым и синим цветами (по пять битов). При этом всего получается $2^5 = 32$ оттенка каждого цвета, т.е. палитра, состоящая из $32 \times 32 \times 32 = 32\,768$ цветов.

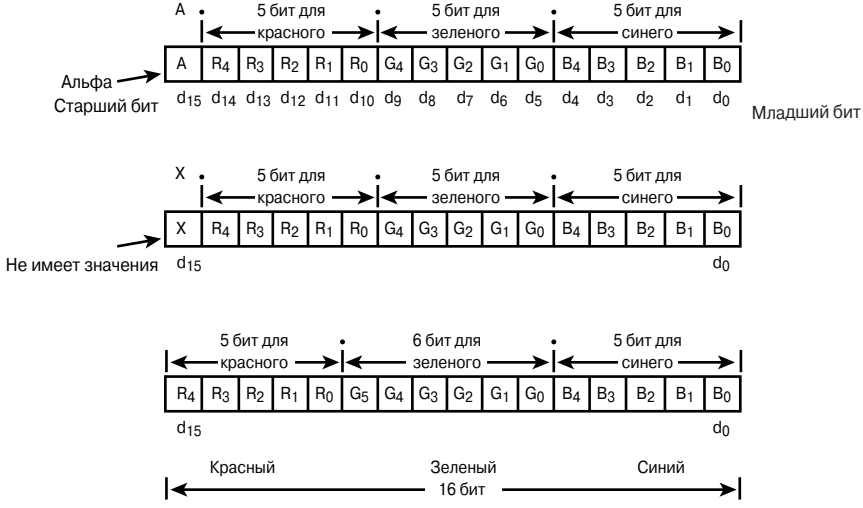


Рис. 7.1. Различные 16-битовые кодировки пикселей в формате RGB

X.5.5.5. Этот режим подобен предыдущему за исключением того, что старший бит остается неиспользованным и в нем может храниться что угодно. Диапазон каждого основного цвета (красного, зеленого и синего), как и в предыдущей кодировке, состоит из 32 оттенков, с общим количеством $32 \times 32 \times 32 = 32\,768$ цветов.

5.6.5. Этот режим применяется чаще всего. В нем цвета определяются с помощью всех 16 бит, входящих в переменную типа `WORD`. Как можно догадаться, в этом формате для красного цвета выделяется 5 бит, для зеленого — 6, для синего — 5, что в сумме дает $32 \times 64 \times 32 = 65\,536$ цветов. Может возникнуть вопрос: "Почему зеленому цвету отдается предпочтение?" Дело в том, что глаз человека наиболее чувствителен именно к зеленому, поэтому расширение диапазона именно этого цвета — наиболее логически обоснованный шаг.

Теперь, когда вы ознакомились с битовым кодированием форматов RGB, необходимо выяснить, как их создавать. Эта задача реализуется с помощью обычных операций побитового сдвига и маскирования. Ниже приведены макросы, реализующие эти операции.

```
// Создание значения 16-битового цвета в формате 5.5.5
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + \
                ((r & 31) << 10))

// Создание значения 16-битового цвета в формате 5.6.5
// (преобладание зеленого)
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + \
                ((r & 31) << 11))
```

Из макросов и рис. 7.2 легко понять, что предназначенные для красного цвета биты расположены в старших разрядах цветового значения типа `WORD`, для зеленого — в средних разрядах, а для синего — в младших разрядах. Может показаться, что все наоборот,

потому что персональные компьютеры — эти *остроконечники*[1] — ведут запись данных в порядке возрастания разрядности битов. Однако в данном случае биты записываются в *тупоконечном* формате, что намного удобнее, потому что этот порядок совпадает с форматом RGB, т.е. в направлении от старшего бита к младшему.

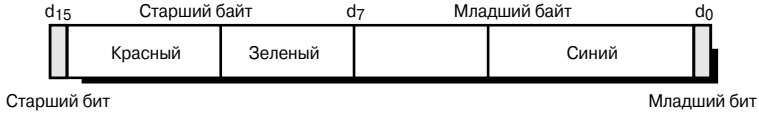


Рис. 7.2. Цветовые значения типа WORD в "тупоконечном" формате

ВНИМАНИЕ

Прежде чем приступить к созданию небольшого демонстрационного примера в 16-битовом режиме, необходимо выяснить один вопрос: как узнать, какой режим используется — 5.5.5 или 5.6.5? Важно научиться получать эту информацию, поскольку режим не подлежит вашему контролю. В DirectDraw можно задать 16-битовый режим, однако кодировка битов полностью зависит от аппаратного обеспечения. Вид кодировки необходимо точно выяснить, иначе зеленый канал может быть полностью разрушен! Чтобы этого не случилось, нужно знать формат пикселей.

Получение информации о формате пикселей

Чтобы выяснить формат пикселей, из которых состоит поверхность, нужно воспользоваться функцией IDIRECTDRAWSURFACE7:GetPixelFormat(), прототип которой приведен ниже.

```
HRESULT GetPixelFormat(LPDDPIXELFORMAT lpDDPixelFormat);
```

Структура DDPIXELFORMAT уже встречалась в предыдущей главе. Нас интересуют такие поля этой структуры:

```
DWORD dwSize;          // Размер структуры (задается программистом)
DWORD dwFlags;         // Флаги, описывающие структуру (табл. 7.1)
DWORD dwRGBBitCount;   // Количество битов для красного,зеленого и синего цветов
```

Перед тем как запрашивать информацию, необходимо задать размер структуры DDPIXELFORMAT в поле dwSize. После вызова к этому размеру поля dwFlags и dwRGBBitCount будут содержать значения информационных флагов, а также количество RGB-битов рассматриваемой поверхности. В табл. 7.1 приведен список некоторых возможных флагов, которые могут использоваться в поле dwFlags.

Таблица 7.1. Возможные значения флагов DDPIXELFORMAT.dwFlags

Значение	*Описание*
DDPF_ALPHA	В этом формате описываются только альфа-значения поверхности
DDPF_ALPHAPIXELS	В этом формате пиксели поверхности содержат информацию об альфа-канале
DDPF_LUMINANCE	В этом формате содержится информация только о яркости или о яркости и прозрачности
DDPF_PALETTEINDEXED1	Поверхность проиндексирована 1-битовыми цветами
DDPF_PALETTEINDEXED2	Поверхность проиндексирована 2-битовыми цветами

[1] Терминология позаимствована автором из романа Свифта "Путешествия Гулливера". — *Прим. перев.*

Значение	*Описание*
DDPF_PALETTEINDEXED4	Поверхность проиндексирована 4-битовыми цветами
DDPF_PALETTEINDEXED8	Поверхность проиндексирована 8-битовыми цветами. Применяется чаще всего
DDPF_PALETTEINDEXEDTO8	Поверхность представлена 1-, 2- или 4-битовыми цветами, проиндексированными для 8-битовой палитры
DDPF_RGB	В структуре формата пикселей достоверны данные по палитре RGB
DDPF_ZBUFFER	В этом формате описываются данные по z-буферу поверхности
DDPF_ZPIXELS	В пикселях поверхности содержится информация по глубине

Примечание. Это далеко не все возможные флаги, особенно для свойств, связанных с трехмерными эффектами. Более подробную информацию можно найти в документации DirectX SDK.

Ниже перечислены наиболее важные поля.

Поле **DDPF_PALETTEINDEXED8** указывает на то, что поверхность построена в 8-битовом режиме с палитрой.

Поле **DDPF_RGB** указывает на то, что поверхность построена в формате RGB и что информацию по формату можно получить, запросив значение dwRGBBitCount.

Таким образом, все, что нужно сделать, — это написать тест, который выглядит примерно так:

```
DDPIXELFORMAT ddpixel; // Для хранения информации
LPDIRECTDRAWSURFACE7 lpdds_primary; // Считаем, что значение корректно

// Очистка структуры
memset(&ddpixel, 0, sizeof(ddpixel));

// Установка размера
ddpixel.dwSize = sizeof(ddpixel);

// Запрос формата пикселей
lpdds_primary->GetPixelFormat(&ddpixel);

// Проверяем, является ли данный режим
// режимом RGB или режимом с палитрой
if (ddpixel.dwFlags & DDPF_RGB)
{
  // Режим RGB;
  // вид режима RGB
  switch(ddpixel.dwRGBBitCount)
  {
    case 15: // Режим 5.5.5
    {
      // Применяется макрос _RGB16BIT555(r,g,b)
    } break;
    case 16: // Режим 5.6.5
    {
    // Применяется макрос _RGB16BIT565(r,g,b)
    } break;
```

```
      case 24: // Режим 8.8.8
      {
      } break;
      case 32: // Режим alpha(8).8.8.8
      {
      } break;
      default: break;
    } // switch

} // if
else
if (ddpixel.dwFlags & DDPF_PALETTEINDEXED8)
{
   // 256-цветный режим с палитрой
} // if
else
{
   // Что-то еще, требуются другие проверки
} // if
```

Довольно просто, не так ли? Истинная мощь функции GetPixelFormat() проявляется, когда видеорежим не задается, а просто создается первичная поверхность в оконном режиме. В этом случае программист не имеет никакого представления о свойствах видеосистемы и должен обязательно их запросить. Иначе глубина цветов, формат пикселей и даже разрешение системы останутся неизвестными.

Теперь, когда вы стали экспертами по 16-битовым режимам, обратимся к демонстрационному примеру. 16-битовое приложение создается очень просто: достаточно вызвать функцию SetDisplayMode() с 16-битовой глубиной цвета. В качестве примера рассмотрим шаги, которые нужно предпринять для создания полноэкранного приложения DirectDraw в 16-битовом цветовом режиме. Вот пример соответствующего кода:

```
LPDIRECTDRAW7 lpdd = NULL;  // Используется  для  запроса directdraw7
DDSURFACEDESC2 ddsd;            // Описание поверхности
LPDIRECTDRAWSURFACE7 lpddsprimary = NULL; // Первичная поверхность
// создание интерфейса IDirectDraw7 и проверка ошибок
if (FAILED(DirectDrawCreateEx(NULL, (void**)&lpdd,
   IID_IDirectDraw7, NULL)))
   return(0);

// Настройка уровня взаимодействия запрошенного режима
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle,
        DDSCL_ALLOWMODEX| DDSCL_FULLSCREEN|
        DDSCL_EXCLUSIVE| DDSCL_ALLOWREBOOT)))
   return(0);

// Установка видеорежима с 16-битовыми цветами
if (FAILED(lpdd->SetDisplayMode(640,480,16,0,0)))
   return(0);

// Создание первичной поверхности
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS;
```

```
// Настройка свойств первичной поверхности
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Создание первичной поверхности
lpdd->CreateSurface(&ddsd,&lpddsprimary,NULL);
```

Вот и все. В этом месте работы программы вы бы увидели черный экран (а возможно, непонятное изображение, если в первичном буфере остались какие-то данные).

Чтобы упростить дальнейшее изложение, предположим, что формат пикселей уже известен, и это 16-битовый режим RGB с форматом 5.6.5. Такое предположение согласуется с изложенным материалом, так как именно этот режим и был установлен ранее. В худшем случае это может оказаться формат 5.5.5. Для вывода пикселя на экран выполните ряд действий.

1. Заблокируйте поверхность. В рассматриваемом примере это означает блокировку первичной поверхности с помощью функции Lock().
2. Создайте значение типа WORD для 16-битового режима RGB. Для этого можно воспользоваться одним из приводившихся ранее макросов или своим собственным кодом. По сути, в функцию, осуществляющую вывод пикселей на экран, передаются значения интенсивностей красного, зеленого и синего цветов. Они должны быть скомбинированы в 16-битовый формат 5.6.5, который нужен для построения первичной поверхности.
3. Запишите пиксель в память. Это означает, что нужно выполнить адресацию первичного буфера с помощью указателя USHORT и записать пиксель в буфер видеопамяти.
4. Разблокируйте первичный буфер, вызвав функцию Unlock().

Ниже приведен код простой функции, предназначенной для вывода 16-битовых пикселей.

```
Void Plot_Pixel16(int x, int y, int red, int green, int blue,
          LPDIRECTDRAWSURFACE7 lpdds)
{
// Эта функция выводит пиксель в 16-битовом режиме;
// она малоэффективна...

DDSURFACEDESC2 ddsd; // Описание поверхности DirectDraw

// Создание цвета WORD
USHORT pixel = _RGB16BIT565(red,green,blue);

// Блокировка видеобуфера
DRAW_INIT_STRUCT(ddsd);

lpdds->Lock(NULL,&ddsd,DDLOCK_WAIT|
       DDLOCK_SURFACEMEMORYPTR,NULL);

// Запись пикселя

// Объявление в памяти поверхности указателя на USHORT
USHORT *video_buffer = ddsd.lpSurface;

// Запись данных
video_buffer[x + y*(ddsd.lPitch >> 1)] = pixel;
```

```
// Разблокирование поверхности
lpdds->Unlock(NULL);

} // Plot_Pixel16
```

Обратите внимание на использование несложного макроса DDRAW_INIT_STRUCT(ddsd), обнуляющего данные структуры и устанавливающего в ней значение поля dwSize. Каждый раз писать код для этих действий утомительно, поэтому удобнее создать такой макрос:

```
#define DDRAW_INIT_STRUCT(ddstruct) { memset(&ddstruct,0, \
    sizeof(ddstruct)); ddstruct.dwSize=sizeof(ddstruct); }
```

Рассмотрим функцию Plot_Pixel16() подробнее. Чтобы вывести на первичную поверхность пиксель с координатами (10, 30) и значениями RGB (255, 0, 0), нужно воспользоваться такой инструкцией:

```
Plot_Pixel16(10,30,     // x, y
       255,0,0,         // RGB
       lpddsprimary); // Поверхность назначения
```

Хотя функция выглядит довольно просто, ее эффективность крайне мала. Чтобы улучшить эту функцию, в ней можно выполнить ряд оптимизаций. Первая проблема состоит в том, что функция каждый раз блокирует поверхность, а затем снимает с нее блокировку, хотя в этом нет никакой необходимости. На некоторых видеокартах операция блокировки-разблокировки может происходить в течение сотен микросекунд, а иногда и дольше. Вывод: в цикле игры следует один раз заблокировать поверхность, выполнить в ней все необходимые манипуляции, а затем снять блокировку (рис. 7.3). При таком подходе не нужно постоянно выполнять блокировку и разблокировку, обнулять память и т.д. Например, заполнение структуры DDSURFACEDESC2, скорее всего, происходит дольше, чем вывод пикселя! Не говоря уж о том, что данная функция не является встраиваемой и непроизводительные затраты ресурсов, необходимые для ее вызова, могут просто "убить" всю производительность.

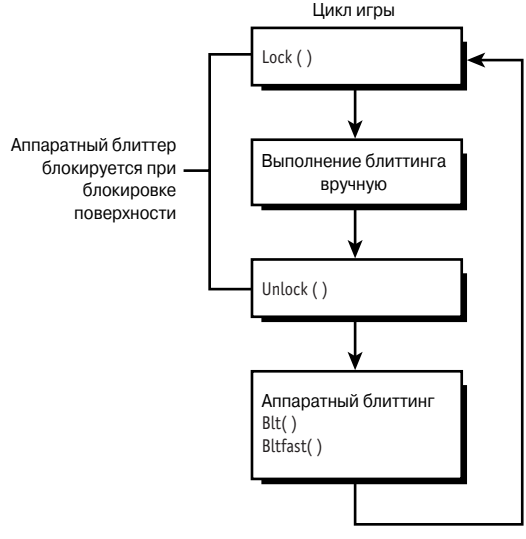


Рис. 7.3. Количество блокировок поверхностей DirectDraw нужно сводить к минимуму

Есть вещи, о которых программисту, занимающемуся разработкой игр, следует помнить всегда. Вот одна из них: вы создаете не текстовый редактор, а игру, поэтому скорость для вас абсолютно критична! Ниже приведена еще одна версия функции, предназначенной для вывода пикселей. В этой новой версии выполнена небольшая оптимизация (хотя все еще остаются резервы для увеличения скорости работы функции минимум в 10 раз).

```
inline void Plot_Pixel_Fast16(int x, int y,
                int red, int green, int blue,
                USHORT *video_buffer, int lpitch)
{
// Эта функция выводит пиксель в 16-битовом режиме;
// предполагается, что в вызывающем модуле поверхность уже
// заблокирована, а также что в нее передан указатель и шаг
// в байтах

// Создание значения цвета типа WORD
USHORT pixel = _RGB16BIT565(red,green,blue);

// Запись данных
video_buffer[x + y*(lpitch >> 1)] = pixel;

} // Plot_Pixel_Fast16
```

Новая версия функции лучше предыдущей, хотя реализация операций умножения и побитового сдвига все еще нуждается в доработке. С помощью пары трюков можно избавиться как от умножения, так и от сдвига. Сдвиг нужен, поскольку lPitch — это ширина памяти, выраженная в байтах. Однако, поскольку в вызывающем модуле поверхность уже заблокирована, а из набора данных, описывающего эту поверхность, уже запрошены указатель и шаг памяти, напрашивается вывод, что в этот же блок имеет смысл добавить еще одно действие, которое состоит в определении переменной lpitch16 типа WORD с шагом, равным 16 битам.

```
int lptch16 = (lpitch >> 1);
```

По сути, lpitch16 — это количество 16-битовых слов, из которых состоит видеострока. С помощью этой новой величины нашу функцию можно еще раз оптимизировать.

```
inline void Plot_Pixel_Faster16(int x, int y,
                int red, int green, int blue,
                USHORT *video_buffer, int lpitch16)
{
// Эта функция выводит пиксель в 16-битовом режиме;
// предполагается, что в вызывающем модуле поверхность уже
// заблокирована, а также что в нее передан указатель и шаг
// в байтах

// Создание значения цвета типа WORD
USHORT pixel = _RGB16BIT565(red,green,blue);

// Запись данных
video_buffer[x + y*lpitch16] = pixel;

} // Plot_Pixel_Faster16
```

Улучшения налицо. Наша функция стала встраиваемой и содержит одинарное умножение, сложение и обращение к памяти. Неплохо, но может быть и лучше. Последняя оптимизация состоит в том, чтобы избавиться от операции умножения путем применения огромной таблицы соответствия. Такая оптимизация может оказаться ненужной, потому что в новейшей архитектуре Pentium X умножение целых чисел сводится к одному циклу, однако на компьютерах с другими процессорами такая оптимизация — реальный путь ускорить работу программ.

Избавиться от умножения можно также с помощью нескольких операций сдвига и сложения. Например, если память работает в идеальном линейном режиме (в строках не имеется свободного места), то в 16-битовом режиме с разрешением 640×480 каждая строка занимает ровно 1 280 байт. В этом случае координату пикселя y нужно умножать на 640, так как при доступе к элементу массива автоматически выполняются арифметические действия с указателем и все, что находится в операторных скобках массива [], умножается на 2 (каждое значение `USHORT WORD` занимает 2 байта). Умножение на 640 можно представить так:

```
y*640 = y*512 + y*128
```

Заметим, что $512 = 2^9$, а $128 = 2^7$. Таким образом, если сдвинуть значение y влево на 9 бит, а затем сложить полученное значение со значением y, сдвинутым на 7 бит влево, результат будет эквивалентен значению y*640:

```
y*640 = y*512 + y*128
      = (y << 9) + (y << 7)
```

Вот и все! Если этот трюк вам не знаком, обратите внимание на рис. 7.4. По сути, сдвиг двоичного числа вправо равнозначен его делению на 2, а сдвиг влево — умножению на 2. Выполнение нескольких сдвигов подряд приводит к их накоплению. Поэтому данное свойство можно применять для ускоренного умножения на числа, представляющие собой степень двойки. Однако если перемножаемые числа не являются степенями двойки, то один из множителей всегда раскладывается на сумму произведений степеней двойки, как было сделано в предыдущем случае. Нужно отметить, что для процессоров Pentium II и выше применение этого трюка не оказывает влияния на скорость работы программы, поскольку они обычно способны перемножать числа за один такт, однако на более старых процессорах или других платформах, таких, как Game Boy Advance, знание описанных выше хитростей всегда оказывается полезным.

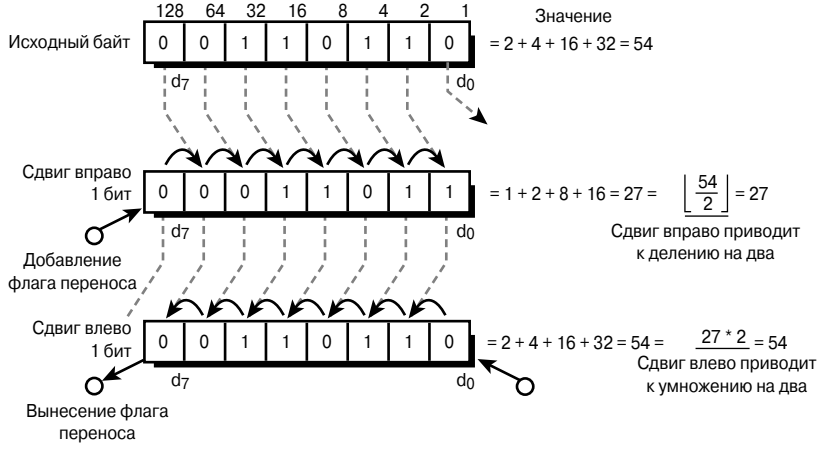


Рис. 7.4. Умножение и деление с помощью бинарных сдвигов

НА ЗАМЕТКУ

В главе 11, “Алгоритмы, структуры данных, управление памятью и многопоточность”, вы ознакомитесь со многими трюками, подобными описанному выше.

В качестве примера вывода пикселя на экран в 16-битовых режимах рассмотрим содержащуюся на компакт-диске программу DEMO7_1.CPP. В этой программе реализован весь материал, изученный к данному моменту. Она попросту выводит на экран случайные пиксели. Взглянув на код программы, можно заметить, что палитра нам не нужна. Этот код составлен с использованием стандартного шаблона T3D Game Engine, поэтому единственное, на что стоит обращать внимание, — это функции Game_Init() и Game_Main(). Ниже приведено содержимое функции Game_Main().

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
   SendMessage(main_window_handle,WM_CLOSE,0,0);

// Вывод на первичную поверхность 1000 случайных пикселей
// и возврат

// Обнуление переменной ddsd и размера набора данных
// (обязательно!)
DDRAW_INIT_STRUCT(ddsd);

// Блокировка первичной поверхности
if (FAILED(lpddsprimary->Lock(NULL, &ddsd,
           DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT,
           NULL)))
   return(0);

// Теперь значения ddsd.lPitch и ddsd.lpSurface корректны

// Несколько дополнительных объявлений, помогающих
// избавиться от преобразований типов
int lpitch16 = (int)(ddsd.lPitch >> 1);
USHORT *video_buffer = (USHORT *)ddsd.lpSurface;

// Вывод на первичную поверхность 1000 пикселей со
// случайным цветом; пиксели отображаются без задержек
for (int index=0; index < 1000; index++)
{
   // Выбор случайного положения и цвета
   // в режиме 640x480x16
   int red   = rand()%256;
   int green = rand()%256;
   int blue  = rand()%256;
   int x = rand()%640;
   int y = rand()%480;

   // Вывод пикселя
   Plot_Pixel_Faster16(x,y,red,green,blue,
               video_buffer,lpitch16);
```

```
} // index

// Разблокировка первичной поверхности
if (FAILED(lpddsprimary->Unlock(NULL)))
   return(0);

// Успешное завершение (или ваш собственный код возврата)
return(1);

} // Game_Main
```

Высокоцветный 24/32-битовый режим

После освоения 16-битового режима 24- и 32-битовый режимы кажутся тривиальными. Начнем с 24-битового режима, потому что он проще 32-битового, что не удивительно. В 24-битовом режиме под каждый канал RGB выделяется ровно по одному байту; таким образом, в этом режиме нет потерь пространства памяти, и в каждом канале получается 256 оттенков, что в сумме дает 256×256×256 = 16.7 миллионов цветов. Биты, в которых хранятся интенсивности красного, зеленого и синего цветов, закодированы так же, как и в 16-битовом режиме, но здесь не нужно беспокоиться, что один из каналов использует больше битов, чем другой.

Так как каждый из трех каналов занимает по одному байту, в пикселе содержится три байта. Это приводит к довольно неприятной адресации (рис. 7.5). Как видно из приведенной ниже 24-битовой версии функции, предназначенной для записи пикселей, процесс записи пикселей в чистом 24-битовом режиме, к сожалению, очень запутан.

```
inline void Plot_Pixel_24(int x, int y,
                          int red, int green, int blue,
                          UCHAR *video_buffer, int lpitch)
{
// Эта функция выводит пиксель в 24-битовом режиме;
// предполагается, что в вызывающем модуле поверхность уже
// заблокирована, а также что в нее передан указатель и шаг
// в байтах

// Выраженная в байтах адресация: 3*x + y*lpitch
// Это адрес младшего байта в синем канале;
// данные записываются в порядке: красный, зеленый, синий
DWORD pixel_addr = (x+x+x) + y*lpitch;

// Запись данных; сначала синий,
video_buffer[pixel_addr]   = blue;

// затем зеленый,
video_buffer[pixel_addr+1] = green;

// наконец красный
video_buffer[pixel_addr+2] = red;

} // Plot_Pixel_24
```

ВНИМАНИЕ

Многие видеокарты из-за ограничений адресации не поддерживают 24-битовый режим. Они поддерживают только 32-битовые цвета, в кодировке которых 8 бит отводится для прозрачности, а остальные 24 бита — для самих цветов. Поэтому демонстрационная программа DEMO7_2.EXE может не работать в вашей системе.

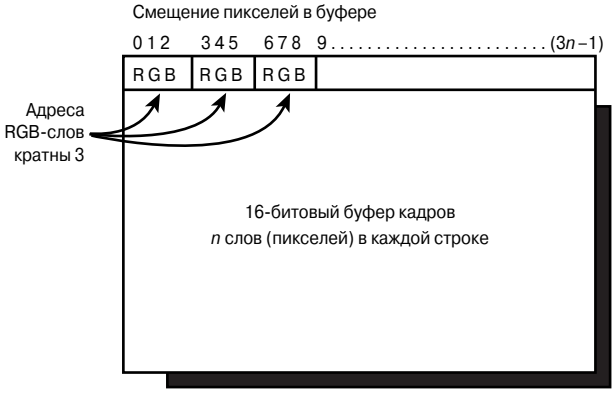


Рис. 7.5. Неудобства трехбайтовой адресации RGB

В число параметров функции входят координаты x и y, цвет в формате RGB, начальный адрес видеобуфера и, наконец, выраженный в байтах шаг памяти. Типы данных, длина которых равняется трем байтам, отсутствуют, поэтому в данной функции адресация видеобуфера использует указатель на байт. Ниже приведен макрос, предназначенный для создания 24-битового слова в формате RGB.

```
// Создание 24-битового значения цвета в формате 8.8.8
#define _RGB24BIT(r,g,b) ((b) + ((g) << 8) + ((r) << 16))
```

В качестве примера реализации 24-битового режима рассмотрите имеющуюся на прилагаемом компакт-диске программу DEMO7_2.CPP. В этой программе выполняется то же, что и в программе DEMO7_1.CPP, но в 24-битовом режиме.

При переходе к 32-битовым цветам расположение битов становится несколько другим (рис. 7.6). В 32-битовом режиме данные пикселей размещаются в двух форматах.

Alpha(8).8.8.8. В этом формате с помощью первых восьми битов передается прозрачность (иногда информация другого вида), а затем для каждого канала, представляющего три основных цвета (красный, зеленый и синий) выделяется по восемь битов. Однако если ведется работа с простыми растрами, то обычно информация относительно прозрачности не принимается во внимание и первые восемь битов просто заполняются восьмерками. Преимущество этого режима состоит в том, что в нем каждый пиксель занимает 32 бита, а это самый быстрый режим адресации памяти для процессоров Pentium.

X(8).8.8.8. Этот режим подобен предыдущему за исключением того, что содержимое восьми старших битов в значении цвета типа WORD не играет роли (однако для надежности им все же лучше присвоить нулевые значения). Вы можете сказать, что этот режим выглядит так же, как и 24-битовый, поэтому непонятно, зачем он нужен. Дело в том, что во многих видеокартах невозможно адресовать данные в трехбайтовых границах, поэтому четвертый байт нужен просто для выравнивания.

Вот макрос для создания 32-битового значения цвета:

```
// Создание 32-битового значения цвета в формате A.8.8.8
// (8 бит, альфа-формат)
#define _RGB32BIT(a,r,g,b) ((b) + ((g) << 8) + \
        ((r) << 16) + ((a) << 24))
```

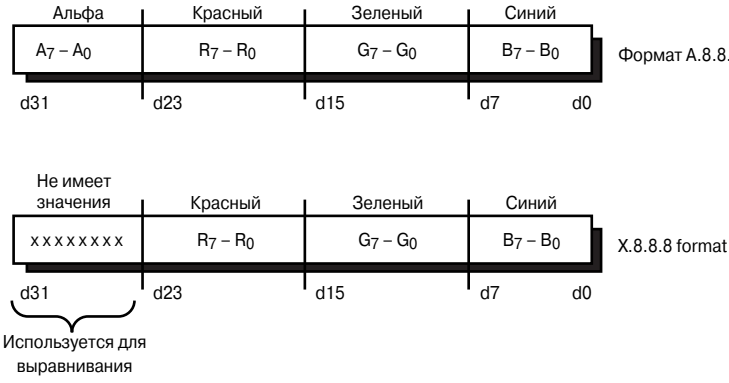


Рис. 7.6. Так выглядят 32-битовые кодировки пикселей в формате RGB

Таким образом все, что вам нужно сделать, — это заменить в функции вывода пикселей старый макрос новым, а также воспользоваться преимуществами четырехбайтового размера пикселей. В результате получим следующий код:

```
inline void Plot_Pixel_32(int x, int y,
          int alpha, int red, int green, int blue,
          UINT *video_buffer, int lpitch32)
{
// Эта функция выводит пиксель в 32-битовом режиме;
// предполагается, что в вызывающем модуле поверхность уже
// заблокирована, а также что в нее передан указатель и шаг
// в байтах

// Создание значения цвета
UINT pixel = _RGB32BIT(alpha,red,green,blue);

// Запись данных
video_buffer[x + y*lpitch32] = pixel;

} // Plot_Pixel_32
```

В приведенном выше коде для вас не должно быть непонятных моментов. Единственное неявное предположение состоит в том, что значение переменной `lpitch32` — это выраженная в байтах величина шага, деленная на четыре, поэтому данное значение принадлежит типу `DWORD`. Приняв это к сведению, рассмотрим работу программы `DEMO7_3.CPP`, которая представляет собой демонстрационный пример реализации вывода пикселей в 32-битовом режиме. Эта программа будет работать на большинстве машин, потому что 32-битовый режим поддерживает большее количество видеокарт, чем 24-битовый.

Теперь все в порядке. Мы уже потратили достаточно времени на освоение высокоцветных режимов, чтобы уметь с ними работать и при необходимости преобразовывать любой 8-битовый цвет. Помните, что нельзя ориентироваться только на тех пользователей, которые имеют процессор Pentium IV 2.0 GHz с ускорителем GeForce III 3D Accelerator. Использование 8-битовых цветов — хороший способ создать работающую программу, после чего можно переходить к 16-битовому или более высоким режимам.

Двойная буферизация

До сих пор мы модифицировали содержимое первичной поверхности; при этом видеоконтроллер отображал каждый кадр непосредственно на экран. Такую методику можно использовать в демонстрационных примерах и для построения статических изображений; однако как быть, если требуется выполнить плавную анимацию? Здесь возникает определенная проблема, суть которой мы постараемся вкратце объяснить. Уже упоминалось, что в процессе компьютерной анимации каждый кадр, как правило, создается во внеэкранном буфере, а затем это изображение очень быстро перебрасывается на видимую поверхность экрана (рис. 7.7).

При этом пользователь не видит, как программист удаляет изображение, генерирует содержимое экрана или выполняет какие-либо иные действия с кадрами. Если копирование внеэкранного изображения на видимую поверхность происходит очень быстро, вы можете работать с частотой не ниже 15 fps, получая достаточно плавную анимацию. Однако современный минимальный стандарт для получения качественной анимации — не менее 30 fps.

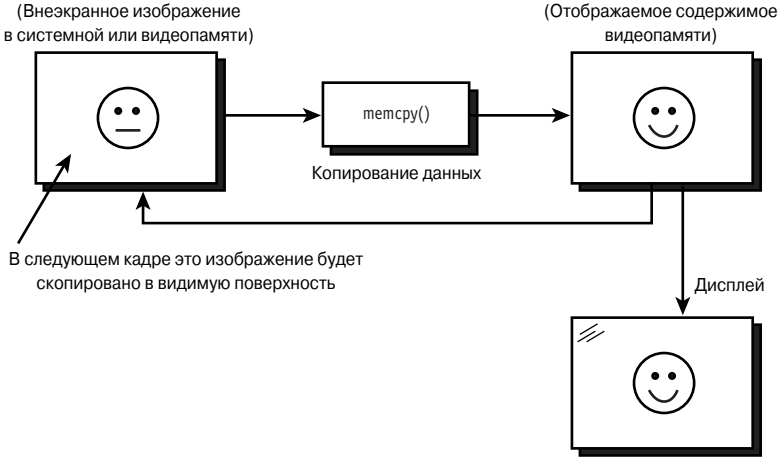


Рис. 7.7. Анимация с помощью двойной буферизации

Процесс построения изображения на внеэкранной поверхности и последующего копирования этого изображения на поверхность дисплея называется *двойной буферизацией* (double buffering). Анимация 99% всех игр выполняется именно так. Однако раньше аппаратных устройств, предназначенных для реализации этого процесса (особенно на компьютерах, работающих под управлением DOS), не было. С появлением DirectX/DirectDraw ситуация явно изменилась.

При наличии аппаратных ускорителей (и достаточного объема видеопамяти на видеокарте) можно реализовать процесс под названием *переключение страниц* (page flipping), подобный процессу двойной буферизации. В основе переключения лежит приблизительно та же идея, что и в двойной буферизации. Различие состоит в том, что сначала на экран выводится одна из двух потенциально видимых поверхностей, а затем аппаратное устройство переключается на вывод другой поверхности. При этом отпадает необходимость копировать изображение, поскольку используется аппаратная система адресации, указывающая устройству преобразования в растровый формат другую начальную позицию памяти. В результате получаем непрерывное переключение страниц и обновление содержимого экрана (отсюда и происхождение термина *переключение страниц*).

Переключение страниц, конечно же, было возможно всегда, и многие разработчики игр пользовались этим приемом, программируя переключение страниц в режимах ModeX (320×200, 320×240, 320×400). Однако при этом приходится прибегать к низкоуровневым и прямым методам. Для выполнения такой задачи обычно приходилось применять ассемблер и программировать видеоконтроллер. Однако при переходе к DirectDraw происходит качественный скачок, речь о котором пойдет в следующем разделе. Я просто хотел дать вам представление о направлении данной главы, прежде чем переходить к дальнейшему изложению подробностей двойной буферизации.

Реализация этого метода тривиальна. Все что нужно сделать — это выделить область памяти с той же геометрией, что и первичная поверхность DirectDraw, записывать в нее каждый кадр анимации, а затем копировать на первичную поверхность дисплея содержимое памяти буфера-двойника. К сожалению, при реализации этой схемы возникают проблемы...

Предположим, мы решили организовать режим DirectDraw 640×480×8. Для этого понадобится выделить двойной буфер размерами 640×480 или линейный массив на 307 200 байт. Следует иметь в виду, что данные отображаются на экран строка за строкой, хотя проблема не в этом. Ниже приведен код, предназначенный для создания двойного буфера.

```
UCHAR *double_buffer = (UCHAR*)malloc(640*480);
```

То же можно выполнить с помощью оператора языка C++ new:

```
UCHAR *double_buffer = new UCHAR[640*480];
```

Какой бы метод мы не применили, в результате будем иметь в области памяти, на которую указывает переменная double_buffer, массив длиной 307 200 байт с линейной адресацией. Чтобы выполнить адресацию одного пикселя с координатами (x,y), следует воспользоваться кодом

```
double_buffer[x + 640*y] = ...
```

Приведенная выше инструкция выглядит вполне разумно, так как на каждую виртуальную строку приходится 640 байт, причем предполагается отображение прямоугольной области, каждая строка которой состоит из 640 байт и в которой содержится 480 строк. Вот тут-то и возникает проблема. Предположим, что заблокирован указатель на первичную поверхность дисплея, который хранится в переменой primary_buffer. Кроме того, предположим, что в процессе блокировки извлечен шаг памяти, значение которого присвоено переменной mempitch (рис. 7.8). Если значение переменной mempitch равно 640, то для копирования double_buffer в primary_buffer можно использовать простой код:

```
memcpy((void*)primary_buffer, (void*)double_buffer,640*480);
```

Почти сразу же double_buffer окажется в первичном буфере.

СЕКРЕТ

Здесь имеется возможность для оптимизации. Обратите внимание, что используется функция memcpy(). Эта функция работает довольно медленно, потому что копирует память побайтово (по крайней мере, в ряде компиляторов). Было бы лучше написать собственную функцию, которая бы копировала за один цикл большее количество данных (32 бита, т.е. значения типа DWORD). Это можно осуществить с помощью встроенного или внешнего ассемблера. Освоив теорию оптимизации, вы поймете, как это сделать. Если учесть, что 32-битовые значения — это максимальная порция данных, которую способен обработать процессор Pentium, предложенная оптимизация является хорошим примером.

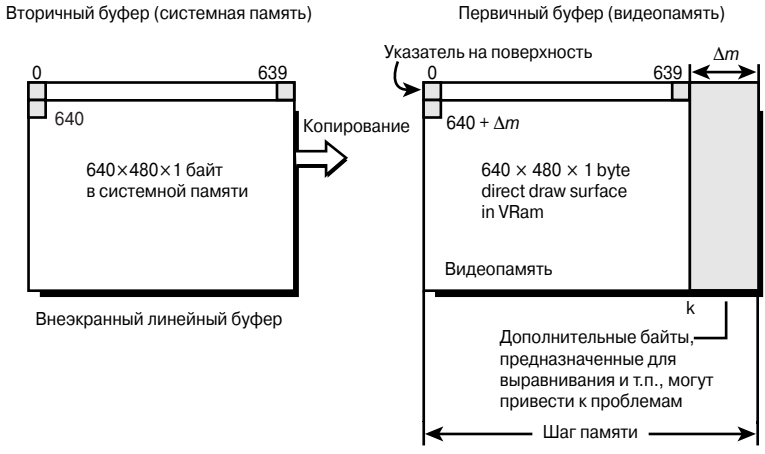


Рис. 7.8. В строках первичной поверхности дисплея может быть дополнительное пространство, что приводит к проблемам адресации

Вроде бы все хорошо, не так ли? Нет, не так! Функция memcpy() в том виде, в котором она была представлена раньше, будет работать только при условии, что значение переменной mempitch (шаг первичной поверхности) в точности равно 640 байт. Это может соответствовать действительности, а может и нет. К сожалению, предыдущая версия функции с использованием memcpy() может с треском провалиться. Лучший способ исправить ситуацию — добавить небольшую функцию, проверяющую, равен ли шаг памяти первичной поверхности значению 640. Если равенство соблюдается, можно применять функцию memcpy(); в противном случае придется осуществлять построчное копирование. При этом программа будет работать немного медленнее, но это лучшее, что можно сделать... Ниже приведен код, в котором реализуется изложенная идея:

```
// Можно ли непосредственно копировать содержимое памяти?
if (mempitch==640)
{
memcpy((void*)primary_buffer, (void*)double_buffer,640*480);
} // if
else
{
// Построчное копирование
for (int y=0; y<480; y++)
   {
   // Копирование очередной строки длиной 640 байт
   memcpy((void*)primary_buffer,(void*)double_buffer,640);

   // Сложная часть — перенос каждого
   // указателя на следующую строку

   // Переход к следующей строке
   primary_buffer+=mempitch;

   // Известно, что нужно произвести сдвиг на 640 байт
   double_buffer+=640;

   } // for y

} // else
```

На рис. 7.9 графически изображена схема, по которой работает приведенный выше код. Как видите, на этот раз пришлось немного потрудиться, а не пользоваться готовыми функциями. Но и этот код позже можно будет оптимизировать, выполняя копирование 4-байтовыми (32-битовыми) блоками.

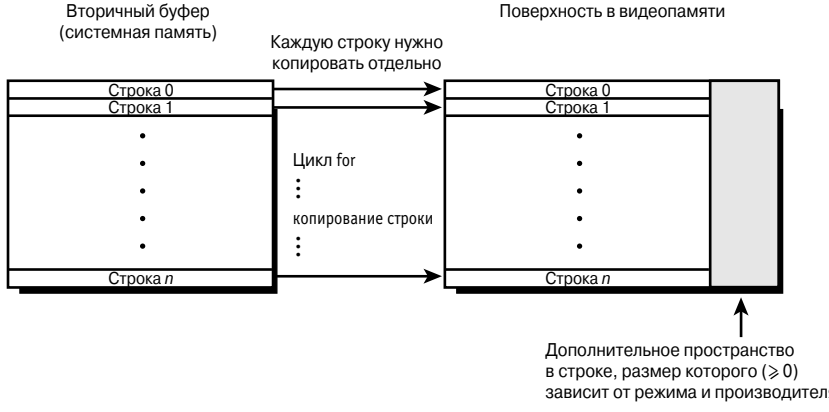


Рис. 7.9. Построчное копирование буфера-двойника

В качестве примера рассмотрим демонстрационную программу, которая заносит в буфер-двойник набор случайных пикселей, а затем копирует содержимое буфера-двойника в первичную поверхность. Программа работает в режиме 640×480×8. Пауза между последовательными копированиями сделана довольно большой, чтобы можно было заметить, что очередное изображение полностью отличается от предыдущего. Программа называется DEMO7_4.CPP и находится на прилагаемом компакт-диске. Компилируя эту программу самостоятельно, не забудьте добавить в проект библиотеку DDRAW.LIB и указать пути к заголовочным файлам в соответствующей папке DirectX. Ниже приведен листинг функции Game_Main(), в котором реализованы перечисленные выше действия.

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

UCHAR *primary_buffer = NULL; // Двойник первичного буфера

// Подстраховка против повторного запуска
if (window_closed)
   return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
   {
   PostMessage(main_window_handle,WM_CLOSE,0,0);
   window_closed = 1;
   } // if

// Обнуление буфера-двойника
memset((void*)double_buffer,0, SCREEN_WIDTH*SCREEN_HEIGHT);
```

```
// Выполнение логики игры...

// Создание в буфере-двойнике очередного кадра;
// вывод 5000 случайных пикселей
for (int index=0; index < 5000; index++)
   {
   int  x  = rand()%SCREEN_WIDTH;
   int  y  = rand()%SCREEN_HEIGHT;
   UCHAR col = rand()%256;
   double_buffer[x+y*SCREEN_WIDTH] = col;
   } // for index

// Копирование буфера-двойника в первичный буфер
DDRAW_INIT_STRUCT(ddsd);

// Блокирование первичной поверхности
lpddsprimary->Lock(NULL,&ddsd, DDLOCK_SURFACEMEMORYPTR
                   | DDLOCK_WAIT,NULL);

// Извлечение указателя на первичную поверхность
primary_buffer = (UCHAR*)ddsd.lpSurface;

// Проверка линейности памяти
if (ddsd.lPitch == SCREEN_WIDTH)
  {
  // Копирование содержимого буфера-двойника
  // в первичный буфер
  memcpy((void *)primary_buffer, (void *)double_buffer,
       SCREEN_WIDTH*SCREEN_HEIGHT);
  } // if
else
  { // Нелинейная

  // Создание копии адресов отправления и назначения
  UCHAR *dest_ptr = primary_buffer;
  UCHAR *src_ptr  = double_buffer;

  // Память нелинейная, построчное копирование
  for (int y=0; y < SCREEN_HEIGHT; y++)
   {
   // Копирование строки
   memcpy((void *)dest_ptr, (void *)src_ptr, SCREEN_WIDTH)

   // Сдвиг указателей на следующую строку
   dest_ptr+=ddsd.lPitch;
   src_ptr +=SCREEN_WIDTH;

   // Приведенный выше код можно заменить более простым:
   // memcpy(&primary_buffer[y*ddsd.lPitch],
   // double_buffer[y*SCREEN_WIDTH], SCREEN_WIDTH);
   // однако он работает медленнее из-за наличия
   // в каждом цикле пересчета и умножения
```

```
  } // for

 } // else

// Разблокировка первичной поверхности
if (FAILED(lpddsprimary->Unlock(NULL)))
  return(0);

// Пауза
Sleep(500);

// Успешное завершение (или ваш собственный код возврата)
return(1);

} // Game_Main
```

Динамика поверхностей

В этой книге уже упоминалось, что можно создавать несколько различных типов поверхностей, но пока вы познакомились только с первичными поверхностями. Перейдем к обсуждению внеэкранных поверхностей. Они бывают двух видов, один из которых называется *задним буфером* (back buffer).

Задние буферы — это применяющиеся в анимационной цепочке поверхности, геометрия и глубина цвета которых совпадает с соответствующими параметрами первичной поверхности. Поверхности, которые находятся в задних буферах, уникальны, поскольку они создаются так же, как и первичная поверхность. Они являются частью цепочки сменяющихся поверхностей. Другими словами, при запросе одной или нескольких вторичных поверхностей, созданных в задних буферах, в DirectDraw по умолчанию предполагается, что они будут использованы в анимационном цикле. На рис. 7.10 показана схема взаимосвязей между первичной поверхностью и вторичными поверхностями в задних буферах.

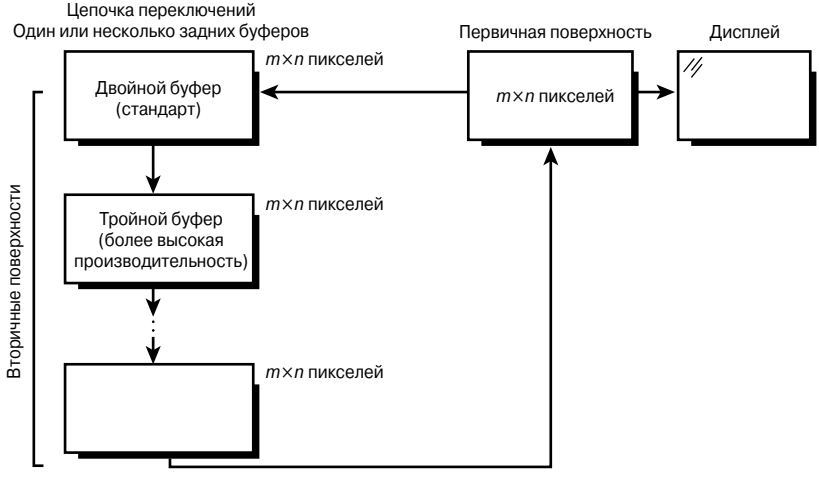


Рис. 7.10. Первичная поверхность и задние буферы

Причина, по которой создается задний буфер, — эмуляция двойной буферизации способом, который больше подходит для DirectDraw. Обычно задний буфер создается в

видеопамяти; в этом случае программа работает очень быстро. Более того, при этом появится возможность осуществлять переключение страниц между задним буфером и первичной поверхностью, что намного быстрее, чем копирование области памяти, которое требуется в схеме двойной буферизации.

Формально задние буферы можно создавать в любом количестве, выстраивая из них цепочку произвольной длины. Однако в некоторый момент придется выйти за рамки видеопамяти и перейти к использованию системной памяти, обмен с которой происходит намного медленнее. Вообще говоря, для создания какого-то режима (скажем, m×n) с насыщенностью цветов, равной одному байту, расходуется память объемом m*n байт (без учета выравнивания шага памяти). Поэтому при наличии еще одного заднего буфера, предназначенного для вторичной поверхности, потребуется в два раза больше памяти, так как задние буферы обладают одинаковой геометрией и глубиной цвета. Таким образом, в этом случае понадобится 2*m*n байт. Наконец, если насыщенность цветов составляет 16 бит, это количество придется увеличить еще в два раза, а если 32 бит — то в целых четыре. Например, объем первичного буфера для режима 640×480×16 бит равен

```
Ширина * Высота * Количество байтов в пикселе =
  640 * 480 * 2 = 614 400 байт.
```

Чтобы создать еще один задний буфер, потребуется в два раза больше памяти; в результате получим

```
614400 * 2 = 1 228 800 байт
```

Почти 1.2 Мбайт видеопамяти! Таким образом, если память видеокарты не превышает 1 Мбайт, о заднем буфере в режиме 640×480×16 можно забыть. В большинстве современных видеокарт содержится не менее 2 Мбайт, поэтому обычно беспокоиться не о чем, хотя все же всегда полезно проверить, какой памятью располагает видеокарта. Для этого можно воспользоваться функцией класса GetCaps (этот вопрос будет рассмотрен в конце главы). В наши дни большинство видеокарт имеют от 8 до 64 Мбайт видеопамяти, однако память многих компьютеров совместно используется несколькими устройствами и в наличии может оказаться всего 2–4 Мбайт.

Чтобы создать первичную поверхность, к которой присоединен задний буфер, необходимо создать конструкцию, которая в DirectDraw называется *сложной поверхностью* (complex surface). Процесс создания сложной поверхности состоит из трех этапов.

1. В поле dwFlags нужно добавить флаг DDSD_BACKBUFFERCOUNT, чтобы указать, что поле dwBackBufferCount структуры DDSURFACEDESC2 будет содержать достоверные данные и что в нем будет содержаться количество задних буферов (в данном случае один).
2. В структуру DDSURFACEDESC2, содержащуюся в поле ddsCaps.dwCaps, необходимо добавить управляющие флаги DDSCAPS_COMPLEX и DDSCAPS_FLIP, которые задают свойства поверхности.
3. Наконец, нужно создать первичную поверхность обычным путем. Далее с помощью функции IDIRECTDRAWSURFACE7::GetAttachedSurface(), прототип которой приведен ниже, необходимо присоединить задний буфер — и дело сделано.

```
HRESULT GetAttachedSurface(LPDDSCAPS2 lpDDSCaps,
  LPDIRECTDRAWSURFACE7 FAR *lplpDDAttachedSurface);
```

Параметр lpDDSCaps — это структура типа DDSCAPS2, в которой содержатся характеристики запрашиваемой поверхности. В данном случае запрашивается задний буфер, который настраивается так:

```
DDSCAPS2 ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
```

Можно не вводить дополнительную переменную, а воспользоваться полем DDSCAPS2 структуры DDSURFACEDESC2:

```
ddsd.ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
```

Ниже приведен код, который предназначен для создания последовательности, состоящей из первичной поверхности и одного заднего буфера.

```
// Предположим, что объект DirectDraw уже создан и
// что проведена вся другая подготовительная работа...

DDSURFACEDESC2 ddsd; // Описание поверхности DirectDraw
LPDIRECTDRAWSURFACE7 lpddsprimary = NULL; // Первичная поверхность
LPDIRECTDRAWSURFACE7 lpddsback    = NULL; // Задний буфер

// Обнуление ddsd и задание размера
DDRAW_INIT_STRUCT(ddsd);

// Подключение полей с достоверными данными
ddsd.dwFlags = DDSD_CAPS| DDSD_BACKBUFFERCOUNT;

// Присвоение полю счетчика задних буферов значения 1
ddsd.dwBackBufferCount = 1;

// Запрос сложной структуры с возможностью переключения
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE|
              DDSCAPS_COMPLEX| DDSCAPS_FLIP;

// Создание первичной поверхности
if (FAILED(lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL)))
  return(0);

// Запрос присоединенной поверхности из первичной поверхности

// Эта строка понадобится для вызова
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;

if (FAILED(lpddsprimary->GetAttachedSurface(&ddsd.ddsCaps,
                           &lpddsback)))
  return(0);
```

С этого момента lpddsprimary указывает на первичную поверхность, которая в данный момент отображается на экране, а lpddsback — на невидимую поверхность в заднем буфере. Изложенная схема в графическом виде показана на рис. 7.11. Чтобы получить доступ к заднему буферу, его можно заблокировать и разблокировать точно так же, как и первичную поверхность.

Таким образом, манипулирование информацией в заднем буфере осуществляется так:

```
// Копирование содержимого буфера-двойника
// в первичную поверхность
DDRAW_INIT_STRUCT(ddsd);

// Блокирование поверхности в заднем буфере
lpddsback->Lock(NULL,&ddsd, DDLOCK_SURFACEMEMORYPTR
                | DDLOCK_WAIT,NULL);
```

```
// Теперь достоверны поля ddsd.lpSurface и ddsd.lPitch;
// выполнение нужных действий...

// Снятие блокировки с буфера, чтобы он стал доступен
// для аппаратных устройств
lpddsback->Unlock(NULL);
```

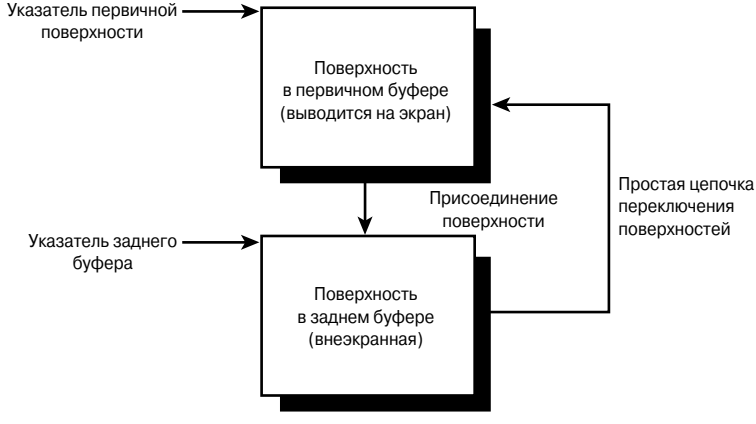


Рис. 7.11. Сложная поверхность

Теперь единственная проблема в том, что вы не умеете переключать поверхности, другими словами, делать поверхность в заднем буфере первичной, осуществляя таким образом анимацию. Продемонстрируем, как это делается.

Переключение страниц

Создав сложную поверхность, состоящую из первичной поверхности и поверхности в заднем буфере, мы готовы к переключению страниц. Стандартный цикл анимации состоит из таких этапов (рис. 7.12).

1. Очистите задний буфер.
2. Сформируйте изображение в заднем буфере.
3. Переключитесь с первичной поверхности на поверхность в заднем буфере.
4. Повторите шаг 1.

На этом этапе могут возникнуть некоторые вопросы. В частности, обмениваются ли первичный и задний буферы своими статусами в результате переключения? Если да, то получается, что в первичном буфере кадры создаются через один? Хотя может показаться, что именно так и происходит, на самом деле все по-другому. В действительности указатели на буферы видеопамяти переключаются аппаратными устройствами, и, с точки зрения программиста и технологии DirectDraw, поверхность в заднем буфере всегда является внеэкранной, а первичная поверхность всегда отображается на экране.

Таким образом, изображение всегда создается в заднем буфере, а затем переключается на первичную поверхность для вывода каждого очередного кадра.

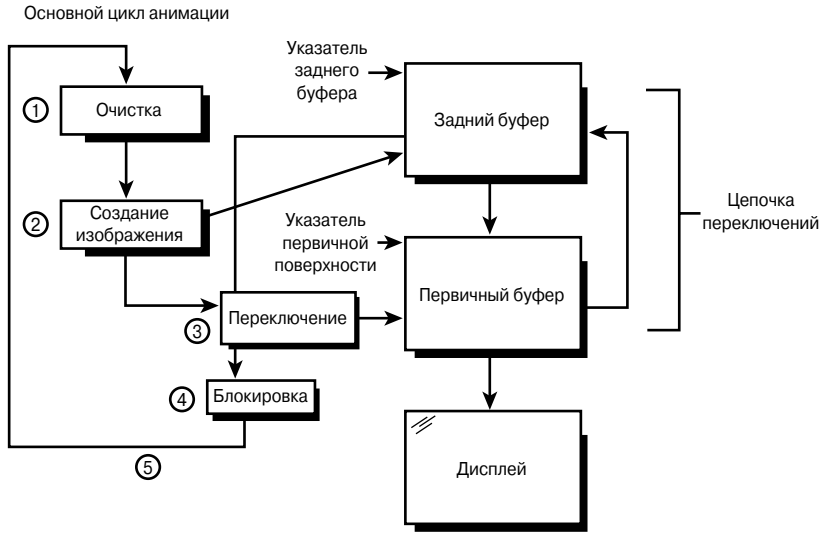


Рис. 7.12. Система анимации с переключением страниц

Чтобы осуществить переключение между первичной поверхностью и следующей в цепочке поверхностью, необходимо воспользоваться функцией IDIRECTDRAWSURFACE7::Flip(), прототип которой приведен ниже.

```
HRESULT Flip(LPDIRECTDRAWSURFACE7 lpDDSurfaceTargetOverride,
                                  // Перезапись поверхности
    DWORD dwFlags);   // Управляющие флаги
```

В случае успешного выполнения эта функция возвращает значение DD_OK; в противном случае возвращается код ошибки.

Функция Flip() имеет простые параметры. lpDDSurfaceTargetOverride — это некоторый дополнительный параметр, которым подменяется последовательная цепочка изображений и с помощью которого происходит переключение между первичной поверхностью и некоторой другой поверхностью, отличной от той, которая присоединена к заднему буферу. В качестве значения этого параметра будет задаваться NULL. А вот параметр dwFlags может нас заинтересовать. В табл. 7.2 перечислены возможные значения этого параметра.

Таблица 7.2. Управляющие флаги функции Flip()

Значение	*Описание*
DDFLIP_INTERVAL2	Переключение после двух вертикальных обратных ходов луча
DDFLIP_INTERVAL3	Переключение после трех вертикальных обратных ходов луча
DDFLIP_INTERVAL4	Переключение после четырех вертикальных обратных ходов луча

(По умолчанию переключение происходит после каждого вертикального обратного хода луча.)

С помощью этих флагов программист указывает, сколько вертикальных обратных ходов луча должно происходить между последовательными переключениями. Если установлен флаг DDFLIP_INTERVAL2, DirectDraw будет осуществлять переключение при каждой второй вертикальной синхронизации; если установлен флаг DDFLIP_INTERVAL3 — при ка-

ждой третьей; если флаг DDFLIP_INTERVAL4 — при каждой четвертой. Эти флаги эффективны только при условии, что в структуре DDCAPS, которая возвращается устройству, установлен флаг DDCAPS2_FLIPINTERVAL.

DDFLIP_NOVSYNC. Если установлен этот флаг, то DirectDraw осуществляет физическое переключение при переходе к очередной строке развертки.

DDFLIP_WAIT. Если при переключении возникают проблемы, то установка этого флага приводит к тому, что аппаратное устройство не станет тут же возвращать код ошибки, а ожидает до тех пор, пока появится возможность выполнить переключение.

СЕКРЕТ

Можно создать сложную поверхность с двумя задними буферами, другими словами, последовательную цепочку, состоящую из трех поверхностей (включая первичную). Такой метод способствует повышению производительности и называется *тройной буферизацией* (triple buffering). Причина повышения производительности очевидна: если задний буфер только один, могут возникнуть задержки, связанные с совместным доступом программы и видеоустройств к заднему буферу. Однако если в цепочке чередования дополнительных буферов два, аппаратным устройствам ждать не приходится. Прелесть тройной буферизации в DirectDraw в ее простоте: вы просто применяете функцию Flip() — и аппаратное устройство переключает поверхности в циклической последовательности. Программист заносит изображения в один и тот же задний буфер, и все происходит по одной и той же схеме.

Обычно устанавливается только флаг DDFLIP_WAIT. Кроме того, функцию Flip() необходимо вызывать как метод из первичной поверхности, а не из заднего буфера. Такая необходимость вызвана тем, что первичная поверхность является "родительской" для поверхности в заднем буфере, а задний буфер — часть родительской цепочки переключений. Как бы там ни было, функция, предназначенная для переключения страниц, вызывается следующим образом:

```
lpddsprimary->Flip(NULL, DDFLIP_WAIT);
```

На тот случай, если функция по каким-то непонятным причинам возвратит ошибку, полезно добавить оператор, проверяющий ее наличие:

```
while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));
```

ВНИМАНИЕ

Перед переключением страниц необходимо снимать блокировку как с поверхности в заденем буфере, так и с первичной поверхности. Поэтому, прежде чем вызвать функцию Flip(), убедитесь, что обе поверхности не заблокированы.

В качестве примера переключения страниц рассмотрим программу DEMO7_5.CPP. Она была получена из программы DEMO7_4.CPP, в которой двойная буферизация заменена переключением страниц. Кроме того, был обновлен код функции Game_Init() — в ней создана сложная поверхность с одним задним буфером. Ниже приведены листинги функций Game_Init() и Game_Main().

```
int Game_Init(void *parms = NULL, int num_parms = 0)
{
// Эта функция вызывается один раз после создания окна
// и перед входом в главный цикл обработки сообщений.
// Здесь выполняются все инициализирующие действия.

// Создание базового интерфейса IDirectDraw 7.0
if (FAILED(DirectDrawCreateEx(NULL, (void**)&lpdd,
                  IID_IDirectDraw7, NULL)))
```

```
   return(0);

// Установка полноэкранного режима
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle,
           DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX |
           DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)))
   return(0);

// Установка видеорежима 640x480x8
if (FAILED(lpdd->SetDisplayMode(SCREEN_WIDTH,
           SCREEN_HEIGHT, SCREEN_BPP,0,0)))
   return(0);


// Обнуление ddsd и задание размера
DDRAW_INIT_STRUCT(ddsd);

// Подключение достоверных полей
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

// Присвоение полю счетчика заднего буфера значения 1,
// для тройной буферизации применяется значение 2
ddsd.dwBackBufferCount = 1;

// Запрос сложной поверхности с возможностью переключения
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                DDSCAPS_COMPLEX | DDSCAPS_FLIP;

// Создание первичной поверхности
if (FAILED(lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL)))
   return(0);

// Запрос присоединенной поверхности из первичной

// Эта строка понадобится при вызове функции
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;

// Получение присоединенной поверхности в заднем буфере
if (FAILED(lpddsprimary->GetAttachedSurface(&ddsd.ddsCaps,
                              &lpddsback)))
   return(0);

// Построение массива данных палитры
for (int color=1; color < 255; color++)
   {
   // Заполнение массива случайными RGB-значениями
   palette[color].peRed   = rand()%256;
   palette[color].peGreen = rand()%256;
   palette[color].peBlue  = rand()%256;

   // Задаем в поле флагов значение PC_NOCOLLAPSE
   palette[color].peFlags = PC_NOCOLLAPSE;
   } // for color
```

```
// Заполнение записей 0 и 255 черным и белым цветами
palette[0].peRed    = 0;
palette[0].peGreen  = 0;
palette[0].peBlue   = 0;
palette[0].peFlags  = PC_NOCOLLAPSE;

palette[255].peRed  = 255;
palette[255].peGreen = 255;
palette[255].peBlue  = 255;
palette[255].peFlags = PC_NOCOLLAPSE;

// Создание объекта-палитры
if (FAILED(lpdd->CreatePalette(DDPCAPS_8BIT |
                    DDPCAPS_ALLOW256 |
                    DDPCAPS_INITIALIZE,
                    palette,&lpddpal, NULL)))
  return(0);

// Присоединение палитры к первичной поверхности
if (FAILED(lpddsprimary->SetPalette(lpddpal)))
  return(0);

// Успешное завершение (или ваш собственный код возврата)
return(1);

} // Game_Init

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

int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

// Проверка повторного запуска
if (window_closed)
  return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
  {
  PostMessage(main_window_handle,WM_CLOSE,0,0);
  window_closed = 1;
  } // if

// Блокировка заднего буфера
DDRAW_INIT_STRUCT(ddsd);
lpddsback->Lock(NULL,&ddsd, DDLOCK_SURFACEMEMORYPTR |
                DDLOCK_WAIT,NULL);

// Указатель-двойник на поверхность в заднем буфере
```

```
UCHAR *back_buffer = (UCHAR *)ddsd.lpSurface;

// Очистка заднего буфера

// Поверка линейности памяти
if (ddsd.lPitch == SCREEN_WIDTH)
   memset(back_buffer,0,SCREEN_WIDTH*SCREEN_HEIGHT);
else
  {
  // Память нелинейная

  // Копирование видеоуказателя
  UCHAR *dest_ptr = back_buffer;

  // Построчная очистка памяти
  for (int y=0; y<SCREEN_HEIGHT; y++)
     {
     // Очистка очередной строки
     memset(dest_ptr,0,SCREEN_WIDTH);

     // Перенос указателя на следующую строку
     dest_ptr+=ddsd.lPitch;

     } // for y

  } // else


// Выполнение логики игры...

// Создание в заднем буфере следующего кадра; обратите
// внимание на необходимость использования переменной
// lpitch, так как поверхность может быть нелинейной

// Вывод 5000 случайных пикселей
for (int index=0; index < 5000; index++)
   {
   int  x  = rand()%SCREEN_WIDTH;
   int  y  = rand()%SCREEN_HEIGHT;
   UCHAR col = rand()%256;
   back_buffer[x+y*ddsd.lPitch] = col;
   } // for index

// Снятие блокировки заднего буфера
if (FAILED(lpddsback->Unlock(NULL)))
   return(0);

// Переключение
while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));

// Пауза
Sleep(500);
```

```
// Успешное завершение (или ваш собственный код возврата)
return(1);

} // Game_Main
```

Обратите внимание на часть кода функции Game_Main(), выделенную жирным шрифтом:

```
// Подстраховка против повторного запуска
if (window_closed)
   return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
  {
  PostMessage(main_window_handle,WM_CLOSE,0,0);
  window_closed = 1;
  } // if
```

СЕКРЕТ

В приведенном выше коде пришлось добавить оператор выхода, так как не исключено, что функция Game_Main() будет вызвана еще раз, уже после удаления соответствующего окна. Это, конечно же, приведет к ошибке, так как DirectDraw должен иметь в своем распоряжении указатель на окно. Поэтому была введена блокирующая переменная (если хотите, бинарный *семафор*), которой после закрытия окна присваивается значение 1, что предотвращает вызов функции Game_Main() в будущем. Это очень важный момент, о котором следовало бы упомянуть при составлении предыдущей программы. Конечно, можно было бы переписать текст и внести в него необходимые дополнения, однако автор пошел другим путем, чтобы продемонстрировать, как легко исправлять ошибки при асинхронном программировании в DirectX/Win32.

Вот и все (или почти все), что следует знать о переключении страниц. Бо́льшая часть работы выполняется с помощью функций DirectDraw; однако напоследок хотелось бы отметить несколько их особенностей. Во-первых, создавая задний буфер, DirectDraw может разместить его не в видеопамяти (если там уже не осталось места), а в системной памяти. В этом случае программист не должен ни о чем беспокоиться; DirectDraw эмулирует все элементы, необходимые для переключения страниц с двойной буферизацией, а также копирует содержимое заднего буфера в первичную поверхность при вызове функции Flip(). Однако при этом все будет происходить медленнее. Но, несмотря ни на что, ваш код будет работать. DirectX — отличная вещь, спасающая от многих проблем!

НА ЗАМЕТКУ

Вообще говоря, создавая первичный и вторичный задние буферы, хотелось бы, чтобы оба они находились в видеопамяти. Первичный буфер всегда располагается в видеопамяти, а вот со вторичным дело обстоит сложнее — он может оказаться и в системной памяти. Всегда следует помнить об ограниченности объема видеопамяти. Иногда программисту выгоднее отказаться от создания заднего буфера в видеопамяти, а вместо этого поместить туда элементы графики игры, чтобы ускорить блиттинг изображений (т.е. передачу изображения из одной области памяти в другую). Аппаратный блиттинг битовых образов из одной области видеопамяти в другую происходит намного быстрее, чем их перемещение из системной памяти в видеопамять. Помещать задний буфер в системную память имеет смысл в тех случаях, когда в игре фигурирует много небольших спрайтов и битовых образов и предполагается их интенсивный блиттинг (и которые вы размещаете в видеопамяти). В этом случае потеря скорости при переключении страниц в схеме двойной буферизации с лихвой компенсируется выигрышем в производительности благодаря тому, что все битовые образы игры содержатся в видеопамяти.

Использование блиттера

Те, кто программировал под DOS, не имели возможности не только окунуться в 32-битовый мир (даже с помощью расширителя DOS), но и воспользоваться аппаратным ускорением для двухмерной или трехмерной графики, если у них не было соответствующего драйвера от производителя или мощной библиотеки, созданной третьей стороной. Аппаратное ускорение получило широкое распространение еще со времени, предшествовавшего появлению игры DOOM, но разработчики игр очень редко его применяли, поскольку оно больше было присуще операционной системе Windows. Однако в DirectX появилась возможность пользоваться всеми преимуществами ускорения — графическими, звуковыми, ввода, сетевыми и т.д. Больше всего радует то, что наконец можно применять аппаратный блиттер для переноса битовых образов и заполнения! Рассмотрим, как это работает.

Обычно создавать битовый образ или заполнять видеоповерхность приходится вручную, пиксель за пикселем. Рассмотрим, например, рис 7.13, на котором изображен битовый образ размером 8×8 с использованием 256 цветов. Пусть нужно скопировать этот образ в видеобуфер или во внеэкранный буфер с размерами 640×480 и линейным шагом и разместить его в точке с координатами (*x*,*y*). Ниже приведен код, с помощью которого выполняются эти действия.

```
UCHAR *video_buffer; // Указатель на видеопамять или
                     // внеэкранную поверхность
UCHAR bitmap[8*8];   // Содержит битовый образ в форме строк

// Неоптимизированное копирование битового образа

// Внешний цикл соответствует отдельным строкам
for (int index_y=0; index_y<8; index_y++)
{
  // Внутренний цикл соответствует пикселям,
  // из которых состоят строки
  for (int index_x=0; index_x<8; index_x++)
  {
    // Копирование писклей без использования прозрачности
    video_buffer[x+index_x + (y+index_y)*640] =
            bitmap[index_x + index_y*8];
  } // for index_x

  } // for index_y
```

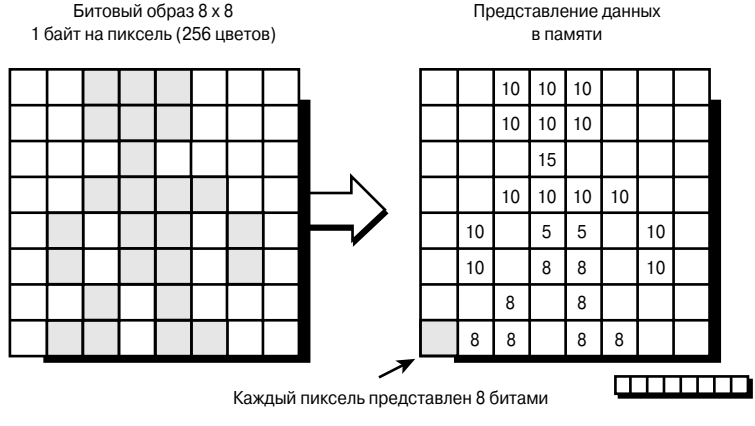


Рис. 7.13. Изображение 8×8 пикселей с использованием 256 цветов

А теперь потратьте несколько минут (или секунд, если вы киборг), чтобы убедиться, что вы понимаете происходящее и способны самостоятельно, не подглядывая, написать подобный код. Изображенная на рис. 7.13 схема поможет вам в этом. По сути, происходит обычное копирование битового образа прямоугольной формы из одной области памяти в другую. Очевидно, с приведенным выше кодом связано несколько проблем и его можно оптимизировать. Сначала поговорим о проблемах.

Проблема 1. Код работает крайне медленно.

Проблема 2. В нем не принимается во внимание прозрачность. Это означает, что игровой объект на черном фоне будет скопирован вместе с фоном. Суть проблемы поясняет рис. 7.14. Чтобы ее решить, нужно добавить дополнительный код.

Что касается оптимизации, то в этом направлении можно выполнить такие действия.

Оптимизация 1. Избавьтесь от всех операций умножения и от большинства операций сложения путем предварительного вычисления начальных адресов копируемого буфера и буфера назначения с последующим увеличением указателей каждого пикселя.

Оптимизация 2. Применяйте операцию заполнения памяти только для непрозрачных пикселей.

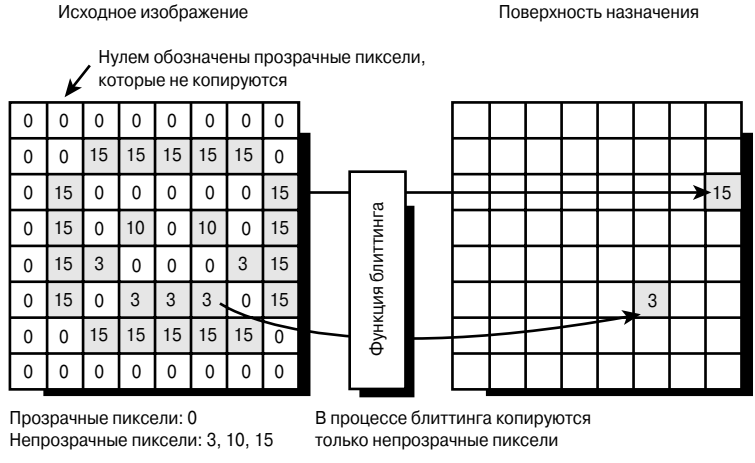


Рис. 7.14. В процессе блиттинга прозрачные пиксели не копируются в поверхность назначения

Приступим к составлению полноценной функции, принимающей во внимание прозрачность (с применением цвета 0) и использующей более оптимальную адресацию, что избавляет от выполнения операций умножения и ускоряет работу. Ниже приведен пример такой функции.

```
void Blit8x8(int x, int y,
        UCHAR *video_buffer,
        UCHAR *bitmap)
{
// Эта функция пересылает изображение в битовый образ,
// находящийся в поверхности назначения с указателем
// video_buffer. Предполагается, что установлен режим
// 640x480x8 с линейным шагом

// Вычисление начального адреса видеобуфера
// video_buffer = video_buffer + (x + y*640)
video_buffer += (x + (y << 9) + (y << 7));
```

```
UCHAR pixel; // Применяется для чтения и записи пикселей

// Основной цикл
for (int index_y=0; index_y < 8; index_y++)
{
  // Внутренний цикл, в котором происходят вычисления
  for (int index_x=0; index_x < 8; index_x++)
  {
    // Копирование пикселей с проверкой их прозрачности
    if (pixel = bitmap[index_x])
        video_buffer[index_x] = pixel;
  } // for index_x

  // Перенос указателей
  bitmap+=8;              // Следующая строка битового образа
  video_buffer+=640;  // Следующая строка видеобуфера
} // for index_y
} // Blit8x8
```

Эта версия функции блиттинга работает во много раз быстрее, чем предыдущая, в которой применялись операции умножения; кроме того, новая версия работает даже с теми битовыми образами, которые содержат прозрачные пиксели! Это упражнение демонстрирует, как с помощью простых изменений можно избавиться от выполнения многих лишних тактов процессора. Но при подсчете циклов оказывается, что новая версия функции все еще достаточно громоздка. Конечно же, определенные непроизводительные затраты являются результатом использования циклов, однако и сам внутренний код функции все еще выглядит небезупречно. Без проверки прозрачности не обойтись, два доступа к элементам массива, запись в память... Вот почему придумали аппаратные ускорители. Использование ускорения позволяет сэкономить циклы процессора для выполнения других действий, например реализации логики игры и ее физики.

Не говоря уже о том, что приведенная выше функция блиттинга просто глупа. В ней жестко закодирован режим 640×480×256, не выполняется отсечение изображения, и она работает только с 8-битовыми изображениями.

Ознакомившись с традиционным способом создания битовых образов, рассмотрим, как заполнять память с помощью блиттера. Затем научимся копировать изображения из одной поверхности в другую. В ходе изучения последующих разделов этой главы с помощью блиттера вы будете рисовать игровые объекты, но до этого необходимо освоить методы работы с ним.

Заполнение памяти с помощью блиттера

Хотя доступ к блиттеру под управлением DirectDraw осуществляется намного проще, чем вручную, блиттер все равно представляет собой достаточно сложное аппаратное устройство. И прежде чем приступить к серьезному использованию какого-то нового видеоустройства, отнюдь не помешает потренироваться на простых задачах. Поэтому рассмотрим, как выполняется очень простое действие — заполнение памяти.

Это действительно просто: некоторая область видеопамяти заполняется определенными значениями. Мы делали это уже неоднократно, блокируя память, а затем вызывая функции memset() или memcpy() для заполнения памяти поверхности. Однако при таком подходе возникают некоторые проблемы. Во-первых, для заполнения памяти используется процессор и в результате в передаче участвует главная шина. Во-вторых, область видеопамяти, в которой хранится поверхность, может не быть полностью линейной, по-

этому заполнение или перемещение придется выполнять построчно. Однако с помощью блиттера заполнение или перемещение участков видеопамяти или поверхностей DirectDraw происходит незамедлительно.

Блиттинг поддерживают две функции, входящие в состав DirectDraw, — IDIRECTDRAWSURFACE7::Blt() и IDIRECTDRAWSURFACE7::BltFast(). Приведем прототипы этих функций:

```
HRESULT Blt(
  LPRECT lpDestRect,                      // Прямоугольник назначения
  LPDIRECTDRAWSURFACE7 lpDDSrcSurface,  // Поверхностьназначения
  LPRECT   lpSrcRect,                     // Прямоугольник-источник
  DWORD    dwFlags,                       // Управляющие флаги
  LPDDBLTFX lpDDBltFx);                   // Специальный параметр
```

Ниже приведено описание параметров, а на рис. 7.15 они представлены на графической схеме.

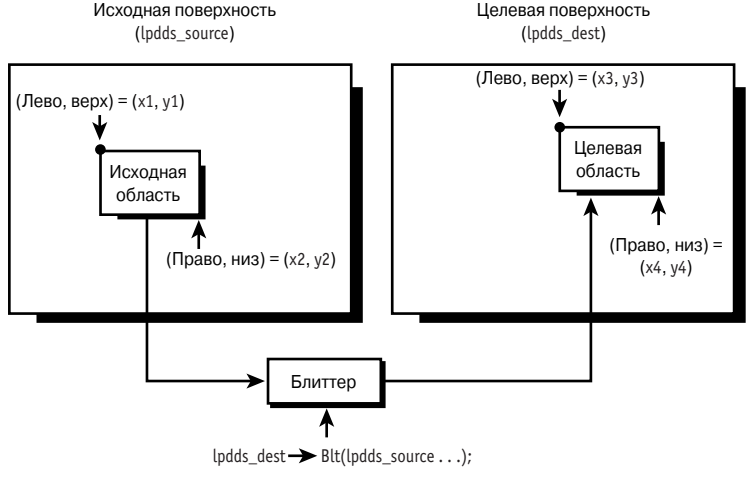


Рис. 7.15. Блиттинг изображения из источника в место назначения

lpDestRect — это адрес структуры RECT, в которой задаются верхняя левая и нижняя правая угловые точки прямоугольника, пересылаемого в поверхность назначения. Если значение этого параметра NULL, то используется вся поверхность назначения.

lpDDSrcSurface — адрес интерфейса IDIRECTDRAWSURFACE7 поверхности DirectDraw, которая будет использоваться в качестве источника блиттинга.

lpSrcRect — адрес структуры RECT, в которой задаются верхняя левая и нижняя правая угловые точки прямоугольника, пересылаемого из поверхности-источника.

dwFlags — определяет достоверные члены следующего параметра, а именно указателя на структуру DDBLTFX, с помощью которой можно управлять поведением объектов (например, изменением их масштаба или угла поворота), а также информацией о цветовых ключах. В табл. 7.3 приведены флаги, которые могут использоваться в параметре dwFlags.

lpDDBltFx — структура, в которой содержится информация, связанная с запрашиваемым блиттингом. Эта структура данных выглядит следующим образом:

```
typedef struct _DDBLTFX
{
DWORD dwSize;  // Размер данной структуры в байтах
DWORD dwDDFX;  // Тип эффекта блиттера
```

```
DWORD dwROP;                  // Поддерживаемые параметры растра Win32
DWORD dwDDROP;                // Поддерживаемые параметры растра DirectDraw
DWORD dwRotationAngle;         // Угол вращения
DWORD dwZBufferOpCode;         // Поля z-буфера (для опытных
                               // программистов)
DWORD dwZBufferLow;            // Этот и последующие параметры
DWORD dwZBufferHigh;           // предназначены для тонкой
DWORD dwZBufferBaseDest;       // настройки блиттера
DWORD dwZDestConstBitDepth;  // опытными программистами
union
{
  DWORD          dwZDestConst;
  LPDIRECTDRAWSURFACE lpDDSZBufferDest;
};
DWORD dwZSrcConstBitDepth;
union
{
  DWORD          dwZSrcConst;
  LPDIRECTDRAWSURFACE lpDDSZBufferSrc;
};
DWORD dwAlphaEdgeBlendBitDepth;
DWORD dwAlphaEdgeBlend;
DWORD dwReserved;
DWORD dwAlphaDestConstBitDepth;
union
{
  DWORD dwAlphaDestConst;
  LPDIRECTDRAWSURFACE lpDDSAlphaDest;
};
DWORD dwAlphaSrcConstBitDepth;
union
{
  DWORD dwAlphaSrcConst;
  LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
};
union // Важные члены
{
  DWORD dwFillColor;  // Цвет, применяющийся для заполнения
  DWORD dwFillDepth; // Глубина заполнения (дополнительный
                     // параметр)
  DWORD dwFillPixel;  // Цвет для заполнения в формате RGB
                     // (альфа)
  LPDIRECTDRAWSURFACE lpDDSPattern;
};
// Важные члены
DDCOLORKEY ddckDestColorkey; // Цветовой ключ заполнения
DDCOLORKEY ddckSrcColorkey;  // Цветовой ключ источника
} DDBLTFX,FAR* LPDDBLTFX;
```

Наиболее важные поля выделены жирным шрифтом.

Таблица 7.3. Управляющие флаги функции Blt()

Значение	*Описание*
Общие флаги	
DDBLT_COLORFILL	Использует значение члена dwFillColor структуры DDBLTFX в качестве цвета в формате RGB, заполняющего прямоугольник-адресат в поверхности назначения
DDBLT_DDFX	Использует значение члена dwDDFX структуры DDBLTFX для определения эффектов, которые будут использованы в процессе блиттинга
DDBLT_DDROPS	Использует значение члена dwDDROP структуры DDBLTFX для определения растровых операций (raster operations — ROP), не входящих в состав Win32 API
DDBLT_DEPTHFILL	Использует значение члена dwFillDepth структуры DDBLTFX в качестве глубины, применяющейся для заполнения прямоугольника-адресата в поверхности z-буфера назначения
DDBLT_KEYDESTOVERRIDE	Использует значение члена dwckDestColorkey структуры DDBLTFX в качестве цветового ключа поверхности назначения
DDBLT_KEYSRCOVERRIDE	Использует значение члена dwckSrcColorkey структуры DDBLTFX в качестве цветового ключа исходной поверхности
DDBLT_ROP	Использует значение члена dwROP структуры DDBLTFX для применяющихся в процессе блиттинга ROP. Эти ROP совпадают с теми, которые определены в Win32 API
DDBLT_ROTATIONANGLE	Использует значение члена dwRotationAngle структуры DDBLTFX в качестве угла вращения (выраженного в 1/100 долях градуса) на поверхности. Этот флаг работает только при поддержке аппаратного обеспечения. Уровень эмуляции аппаратного обеспечения (HEL) не способен выполнять вращение
Флаги цветовых ключей	
DDBLT_KEYDEST	Использует значение цветового ключа, ассоциированного с поверхностью назначения
DDBLT_KEYSRC	Использует значение цветового ключа, ассоциированного с поверхностью-источником
Флаги образа действий	
DDBLT_ASYNC	Блиттинг выполняется асинхронно, однако в порядке получения, т.е. по принципу “первым вошел — первым вышел” (FIFO). Если в устройствах FIFO нет места, вызов дает сбой. Быстро, но рискованно; чтобы надлежащим образом использовать этот флаг, нужно проверять наличие ошибок
DDBLT_WAIT	Если блиттер занят, он не возвращает ошибку DDERR_WASSTILLDRAWING, а ждет, пока появится возможность выполнить блиттинг

Примечание. Важные и часто применяемые поля выделены жирным шрифтом.

Если вы уже сходите с ума от обилия параметров — не удивляйтесь, потому что с автором происходит то же самое. Теперь рассмотрим функцию BltFast().

```
HRESULT BltFast(
  DWORD  dwX,        // Координата x области назначения
  DWORD  dwY,        // Координата y области назначения
  LPDIRECTDRAWSURFACE7 lpDDSrcSurface,
                     // Исходная поверхность
  LPRECT lpSrcRect,  // Исходный прямоугольник
  DWORD  dwTrans); // Тип передачи
```

Параметры dwX и dwY — координаты (x, y) той области поверхности, в которую будет выполнен блиттинг.

lpDDSrcSurface — адрес интерфейса IDIRECTDRAWSURFACE7, применяющегося для поверхности DirectDraw, которая является источником блиттинга.

lpSrcRect — адрес исходного прямоугольника; в нем задаются верхний левый и нижний правый углы прямоугольника, блиттинг которого осуществляется из исходной поверхности.

dwTrans — тип операции блиттинга. Возможные значения этого параметра приведены в табл. 7.4.

Таблица 7.4. Управляющие флаги функции BltFast()

Значение	*Описание*
DDBLTFAST_SRCCOLORKEY	Блиттинг выполняется с учетом прозрачности и с применением исходного цветового ключа
DDBLTFAST_DESTCOLORKEY	Блиттинг выполняется с учетом прозрачности и с применением цветового ключа назначения
DDBLTFAST_NOCOLORKEY	Обычный блиттинг без учета прозрачности. На некоторых аппаратных устройствах может работать быстрее; работает быстрее на уровне эмуляции аппаратных средств
DDBLTFAST_WAIT	Если блиттер занят, то установка этого флага предотвращает отправку сообщения DDERR_WASSTILLDRAWING. Выполнение функции BltFast() возобновляется как только появляется возможность блиттинга или отменяется, если возникает серьезная ошибка

Примечание. Важные и часто применяемые поля выделены жирным шрифтом.

Сразу же возникает вопрос: "Зачем нужны две разные функции блиттинга?" Ответ ясен из вида самих функций: Blt() — это полнофункциональная модель с большим набором параметров, а BltFast() попроще, однако и параметров у нее поменьше. Кроме того, в функции Blt() есть возможность применять отсечение DirectDraw, а в BltFast() такой возможности нет. Это означает, что на уровне эмуляции аппаратных средств функция BltFast() работает примерно на 10% быстрее, чем Blt(). Функция BltFast() может быть более быстрой даже при работе с аппаратными устройствами (если эти устройства не очень высокого качества). Таким образом, функцию Blt() нужно применять в ситуациях, когда требуется отсечение, а BltFast() — когда отсечения не требуется.

Продемонстрируем процедуру заполнения поверхности с помощью функции Blt(). Эта операция будет достаточно простой по причине отсутствия исходной поверхности (есть только поверхность назначения). В результате многие параметры будут равны значению NULL. Чтобы заполнить память, выполните такие действия.

1. Поместите в поле dwFillColor структуры DDBLTFX индекс или код RGB-цвета, которым следует заполнить поверхность.
2. Заполните структуру RECT параметрами области, которую следует перенести в поверхность назначения.

3. Вызовите из указателя интерфейса IDIRECTDRAWSURFACE7, соответствующего поверхности назначения, функцию Blt() с управляющими флагами DDBLT_COLORFILL|DDBLT_WAIT. Очень важно вызывать функции Blt() и BltFast() из поверхности назначения, а не из исходной поверхности!

Ниже приведен код, предназначенный для заполнения 8-битовой области памяти определенным цветом.

```
DDBLTFX ddbltfx;  // fx-структура блиттера
RECT dest_rect;   // Здесь хранится адрес
          // прямоугольника назначения
// Сначала инициализируем структуру DDBLTFX
DDRAW_INIT_STRUCT(ddbltfx);

// Присвоим слову цвета требуемое значение.
// Предполагается, что установлен 8-битовый режим,
// поэтому будет использован цветовой индекс в диапазоне
// 0-255. Если бы был установлен 16/24/32-битовый режим,
// значение типа WORD следовало бы заполнить кодом RGB.

ddbltfx.dwFillColor = color_index; // (Или код RGB для
                                   // 16-битовых и более высоких режимов)

// Зададим структуру RECT, чтобы заполнить область поверхности
// назначения от точки (x1,y1) до точки (x2,y2)
dest_rect.left   = x1;
dest_rect.top    = y1;
dest_rect.right  = x2;
dest_rect.bottom = y2;

// Вызов блиттера
lpddsprimary->Blt(&dest_rect, // Указатель на область
                              // назначения
        NULL,                 // Указатель на исходную поверхность
        NULL,                 // Указатель на исходную область

        DDBLT_COLORFILL| DDBLT_WAIT,
        // Заполнение цветом; ожидание при необходимости

        &ddbltfx); // Указатель на структуру DDBLTFX,
                   // в которой хранится необходимая информация
```

НА ЗАМЕТКУ

Со всеми структурами типа RECT, передаваемыми большинству функций DirectDraw, связана одна небольшая особенность: как правило, верхняя левая точка включается в прямоугольную область, а нижняя правая не включается. Другими словами, если передается область RECT с угловыми точками (0, 0) и (10, 10), сканируемый прямоугольник будет занимать область с угловыми точками (0, 0) и (9, 9). Это нужно иметь в виду. Чтобы заполнить весь экран в режиме 640x480, в качестве верхней левой нужно задать точку (0, 0), а в качестве нижней правой — точку (640, 480).

Обратите внимание на то, что исходной поверхности и ее прямоугольной области соответствуют значения NULL. Смысл этого в том, что блиттер используется для заполнения одним цветом всего экрана, а не для копирования данных из одной поверхности в другую. Итак, продолжим.

Предыдущий пример предназначен для работы с 8-битовой поверхностью. Для перехода к высокоцветным 16/24/32-битовым режимам достаточно изменить значение ddbltfx.dwFillColor таким образом, чтобы оно отражало значение заполняющего цвета. Можно создать RGB-значение цвета, который должен быть прозрачным.

Например, если дисплей работает в 16-битовом режиме и вам понадобилось заполнить экран зеленым цветом, это можно сделать с помощью такого кода:

```
ddbltfx.dwFillColor = _RGB16BIT565(0,255,0);
```

Все остальное в предыдущем примере для 8-битового режима остается неизменным.

На прилагаемом компакт-диске находится небольшая демонстрационная программа DEMO7_6.CPP, с помощью которой можно познакомиться с работой блиттинга. Эта программа переводит систему в 16-битовый режим 640×480, а затем заполняет различные области экрана случайными цветами. За секунду на экран выполняется блиттинг огромного количества цветных прямоугольников. Попробуйте выключить свет и предаться созерцанию этого зрелища. Рассмотрим листинг функции Game_Main(), которая почти тривиальна.

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

DDBLTFX ddbltfx; // fx-структура блиттера
RECT dest_rect;  // Адрес прямоугольника назначения

// Подстраховка против повторного запуска
if (window_closed)
   return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))

   PostMessage(main_window_handle,WM_CLOSE,0,0);
   window_closed = 1;
} // if

// Инициализация структуры DDBLTFX
DDRAW_INIT_STRUCT(ddbltfx);

// Присвоим слову цвета случайное значение
ddbltfx.dwFillColor = _RGB16BIT565(rand()%256,
                      rand()%256,
                      rand()%256);

// Выбор случайного прямоугольника
int x1 = rand()%SCREEN_WIDTH;
int y1 = rand()%SCREEN_HEIGHT;
int x2 = rand()%SCREEN_WIDTH;
int y2 = rand()%SCREEN_HEIGHT;

// Зададим структуру RECT для заполнения области между
// угловыми точками (x1,y1) и (x2,y2) на поверхности
// назначения
dest_rect.left  = x1;
```

```
dest_rect.top    = y1;
dest_rect.right  = x2;
dest_rect.bottom = y2;

// Вызов блиттера
if (FAILED(lpddsprimary->Blt(
       &dest_rect, // Указатель на область назначения
       NULL,       // Указатель на исходную поверхность
       NULL,       // Указатель на исходную область

       DDBLT_COLORFILL | DDBLT_WAIT,
       // Заполнение цветом; ожидание при необходимости

       &ddbltfx)))
       // Указатель на структуру DDBLTFX,
       // в которой хранится необходимая информация
  return(0);

// Успешное завершение (или ваш собственный код возврата)
return(1);

} // Game_Main
```

Теперь, когда вы научились с помощью блиттера выполнять заполнение, продемонстрируем, как с его помощью копировать данные из одной поверхности в другую. Вот где проявляется истинная мощь блиттера!

Копирование изображений из одной поверхности в другую

Суть работы блиттера заключается в копировании битового образа прямоугольной формы из исходной области памяти в область-адресат. При этом копироваться может как весь экран, так и небольшой битовый образ, которым представлен игровой объект. Фактически вы уже обладаете всеми необходимыми знаниями, хотя, возможно, и не осознаете этого. Если в приведенный выше демонстрационный пример внести некоторые изменения, он станет пригоден для копирования данных из одной поверхности в другую.

В процессе использования функции Blt() мы задаем исходную прямоугольную область и исходную поверхность, а также прямоугольную область и поверхность назначения, т.е. объекты, участвующие в операции блиттинга. Впоследствии блиттер скопирует пиксели из исходной области в область назначения. В роли исходной поверхности и поверхности назначения может выступать одна и та же поверхность, хотя обычно они различны. Вообще говоря, в предыдущем предложении сформулирована идея, которая является основой большинства спрайтовых процессоров. (*Спрайт* (sprite) — это участвующее в игре растровое изображение, перемещающееся по экрану.)

К этому времени вы уже научились создавать первичную поверхность, а также вторичную, выступающую в роли заднего буфера. Однако вы еще не умеете создавать обычные внеэкранные поверхности, не связанные с первичной поверхностью. Не умея их создавать, невозможно выполнять блиттинг. Таким образом, отложим на время описание общего случая блиттинга любой поверхности в первичную и рассмотрим, как выполняется блиттинг заднего буфера в первичную поверхность.

Все, что нужно сделать, чтобы выполнить блиттинг из любой из двух поверхностей (например, из заднего буфера в первичную поверхность), — это правильно задать участвующие в блиттинге области и вызвать функцию Blt() с верно установленными параметрами (см. рис. 7.15). Например, пусть нужно скопировать расположенную на исходной поверхности (в данном случае в заднем буфере) прямоугольную область с угловыми точками (x1, y1) и (x2, y2) в область с угловыми точками (x3, y3) и (x4, y4), расположенную на поверхности назначения (в данном случае на первичной поверхности). Ниже приведен код, с помощью которого выполняется это действие.

```
RECT source_rect,  // Хранит исходную область
   dest_rect;      // Хранит область назначения

// Настройка структуры RECT с
// угловыми точками (x1,y1) и (x2,y2)
source_rect.left  = x1;
source_rect.top   = y1;
source_rect.right = x2;
source_rect.bottom = y2;

// Настройка структуры RECT с
// угловыми точками (x3,y3) и (x4,y4)
dest_rect.left  = x3;
dest_rect.top   = y3;
dest_rect.right = x4;
dest_rect.bottom = y4;

// Вызов блиттера
lpddsprimary->Blt(
      &dest_rect,     // Указатель на область назначения
      lpddsback,      // Указатель на исходную поверхность
      &source_rect,   // Указатель на исходную область
      DDBLT_WAIT,     // Управляющие флаги
      NULL);          // Указатель на структуру DDBLTFX
                      // с необходимой информацией
```

Все очень просто, не так ли? Конечно же, некоторые вопросы, такие, как отсечение и прозрачность, остались незатронутыми. Давайте сначала поговорим об отсечении. Рассмотрим рис. 7.16, на котором приведен битовый образ, перемещаемый по поверхности с учетом отсечения и без учета отсечения. Очевидно, что блиттинг без отсечения приводит к проблемам, если битовый образ начинает выходить за пределы прямоугольника, которым ограничена поверхность назначения. Это может стать причиной перезаписи памяти и других подобных неприятностей. В связи с этим в DirectDraw встроена поддержка отсечения, осуществляемого с помощью интерфейса IDirectDrawClipper. Если программист сам пишет код для растеризации битового образа, как это было сделано в примере Blit8x8(), в него всегда можно добавить и код для отсечения, хотя это замедлит работу программы. Вторым вопросом, связанным с блиттингом, является прозрачность.

Битовый образ всегда создается в виде прямоугольной матрицы пикселей. Однако в процессе блиттинга нужно копировать не все пиксели. Во многих случаях выбирается цвет (черный, синий, зеленый или какой-то другой), который является прозрачным и не копируется (такой прием был применен в примере Blit8x8()). В технологии DirectDraw, как уже упоминалось, такая возможность поддерживается и имеет название *цветовой ключ* (color key).

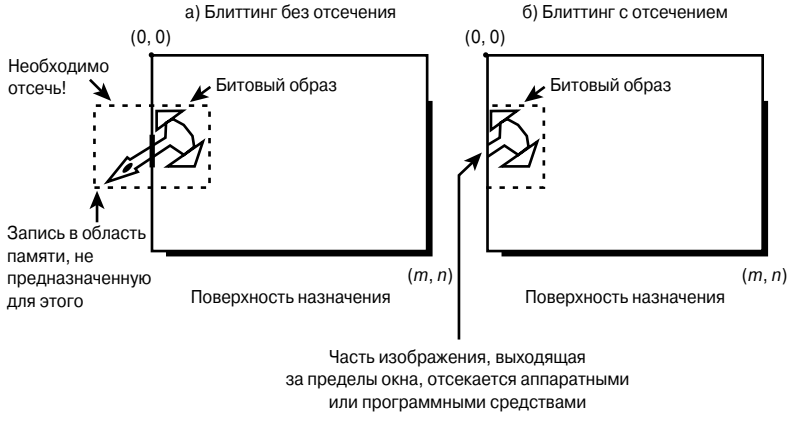


Рис. 7.16. Проблема отсечения битовых образов

Перед тем как перейти к отсечению, ознакомимся с демонстрационной программой, в которой осуществляется блиттинг из заднего буфера в первичную поверхность. Эта программа содержится на прилагаемом компакт-диске в файле DEMO7_7.CPP. Единственная проблема в данной программе заключается в том, что мы пока не умеем загружать битовые образы с диска, поэтому не можем выполнить действительно красочный блиттинг. В рассматриваемом примере выполняются следующие действия. В заднем буфере в 16-битовом цветовом режиме создаются зеленые прямоугольники разной яркости — от самых ярких до почти черных, которые затем используются в качестве исходных данных. На экране отображается множество прямоугольников различных оттенков зеленого цвета со случайным образом выбранными границами; все эти прямоугольники с большой скоростью копируются в первичную поверхность. Ниже приведен исходный код функции Game_Main() из рассматриваемой программы.

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

RECT source_rect,  // Исходный прямоугольник
   dest_rect;      // Прямоугольник назначения

// Подстраховка против повторного запуска
if (window_closed)
  return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
{
  PostMessage(main_window_handle,WM_CLOSE,0,0);
  window_closed = 1;
} // if

// Выбор случайного исходного прямоугольника
int x1 = rand()%SCREEN_WIDTH;
int y1 = rand()%SCREEN_HEIGHT;
```

```
int x2 = rand()%SCREEN_WIDTH;
int y2 = rand()%SCREEN_HEIGHT;

// Выбор случайного прямоугольника назначения
int x3 = rand()%SCREEN_WIDTH;
int y3 = rand()%SCREEN_HEIGHT;
int x4 = rand()%SCREEN_WIDTH;
int y4 = rand()%SCREEN_HEIGHT;

// Задание структуры RECT для заполнения области между
// угловыми точками (x1,y1) и (x2,y2) на исходной
// поверхности
source_rect.left   = x1;
source_rect.top    = y1;
source_rect.right  = x2;
source_rect.bottom = y2;

// Задание структуры RECT для заполнения области между
// угловыми точками (x3,y3) и (x4,y4) на поверхности
// назначения
dest_rect.left   = x3;
dest_rect.top    = y3;
dest_rect.right  = x4;
dest_rect.bottom = y4;

// Вызов блиттера
if (FAILED(lpddsprimary->Blt(
      &dest_rect,      // Область назначения
      lpddsback,       // Исходная поверхность
      &source_rect,  // Исходная область
      DDBLT_WAIT,   // Управляющие флаги
      NULL)))          // Указатель на структуру DDBLTFX,
      // в которой хранится необходимая информация
  return(0);

// Успешное завершение (или ваш код возврата)
return(1);

} // Game_Main
```

Кроме того, в функции Game_Init() есть немного кода на встроенном ассемблере. С помощью этого кода осуществляется заполнение области видеобуфера 32-битовыми значениями типа DWORD, состоящими из двух 16-битовых пикселей в формате RGB; 8-битовое заполнение происходило бы медленнее. Вот этот код:

```
_asm
{
  CLD               ; Установка флага направления
  MOV EAX, color          ; Установка цвета
  MOV ECX, (SCREEN_WIDTH/2) ; Количество значений типа
                    ; DWORDS
  MOV EDI, video_buffer    ; Адреса строк с
                    ; передаваемыми данными
  REP STOSD             ; Цикл пересылки
} // asm
```

По сути, в приведенном выше коде реализован цикл, который на языке C/C++ можно было бы записать так:

```
for(DWORD ecx = 0, DWORD *edi = video_buffer;
  ecx < (SCREEN_WIDTH/2); ecx++)
  edi[ecx] = color;
```

Если вы не знаете ассемблер, не стоит беспокоиться. Этот язык иногда будет использоваться мною для реализации некоторых простых действий, но не более того. Кроме того, это хорошая возможность попрактиковаться в использовании встроенного ассемблера, что поможет вам повысить свой уровень.

В качестве упражнения напишите программу, работающую только с первичной поверхностью. Для этого достаточно удалить код, имеющий отношение к заднему буферу, создать изображение на первичной поверхности, а затем запустить блиттер, в котором в качестве исходной поверхности и поверхности назначения выступала бы одна и та же поверхность. Посмотрим, что получится...

Основные принципы работы с отсечением

В этой книге мы будем неоднократно возвращаться к отсечению поверхностей. Отсечение пикселей, отсечение битовых образов, отсечение двухмерных изображений... наверняка можно придумать что-то еще. Однако в данный момент мы изучаем технологию DirectDraw, поэтому сосредоточим внимание на отсечении пикселей и битовых образов. Это облегчит изучение дальнейшего материала, который, несомненно, станет очень сложным при переходе к трехмерной графике.

В общих словах отсечение можно определить как "предотвращение вывода пикселей или других элементов изображения, которые выходят за рамки смотровой области или окна". Игровая программа, созданная в технологии DirectDraw, должна делать то же самое, что делает Windows, отсекая все выходящее за пределы клиентской области окна приложения.

Если программист разработал графический процессор, с помощью которого можно выводить отдельные пиксели, линии и создавать битовые образы, то он должен реализовать и алгоритмы отсечения. Однако если речь идет о битовых образах в виде поверхностей DirectDraw, то тут весьма полезной может оказаться технология DirectDraw.

Ее помощь состоит в предоставлении отсекателей DirectDraw посредством интерфейса `IDirectDrawClipper`. Все, что нужно сделать, — это создать интерфейс `IDirectDrawClipper`, предоставить ему области, за пределами которых нужно выполнять отсечение, а затем присоединить этот интерфейс к поверхности. После этого в процессе использования функции блиттера `Blt()` будет отсекаться все, что выходит за рамки заданных областей (конечно же, при наличии надлежащего аппаратного обеспечения). Однако сначала рассмотрим, как переписать функцию `Blit8x8()`, чтобы в ней выполнялось отсечение пикселей.

Отсечение пикселей, выходящих за пределы области видимости

Суть проблемы показана на рис. 7.17. Пусть нужно отсечь все пиксели, координаты (x,y) которых находятся за пределами области видимости, ограниченной угловыми точками с координатами (x1,y1) и (x2,y2). Если пиксель попадает в указанную прямоугольную область, он визуализируется, в противном случае — нет. Достаточно просто, не так ли?

Ниже приведен код, реализующий описанное выше действие в 8-битовом режиме 640×480.

```
// Считаем ограничивающий прямоугольник глобальным
int x1,y1,x2,y2; // Эти переменные определены глобально
```

```
...

void Plot_Pixel_Clip8(int x, int y,
                      UCHAR color,
                      UCHAR *video_buffer)
{
  // Проверка, попадает ли пиксель в заданную область
  if (x>=x1 && x<=x2 && y>=y1 && y<=y2)
    video_buffer[x+y*640] = color;

} // if
```

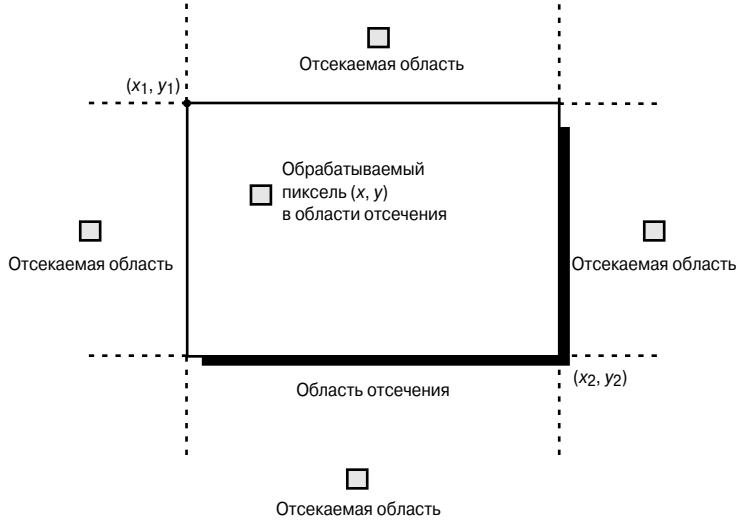


Рис. 7.17. Область отсечения в увеличенном масштабе

Конечно же, в приведенном выше коде остается много возможностей для оптимизации, однако основная идея понятна: создан программный фильтр, отсекающий пиксели на основе значений их координат. Через этот фильтр проходят только те пиксели, координаты которых удовлетворяют условному оператору. Приведенный выше отсекатель является слишком общим. В большинстве случаев одна из угловых точек окна или области видимости имеет координаты (0,0), а другая определяется размерами окна (win_width,win_height). Правда, при этом код практически не упрощается:

```
// Считаем ограничивающий прямоугольник глобальным
int x1,y1,x2,y2; // Эти переменные определены глобально

...

void Plot_Pixel2_Clip8(int x, int y,
                       UCHAR color,
                       UCHAR *video_buffer)
{
  // Проверка, попадает ли пиксель в заданную область
  if (x>=0 && x<=win_width && y>=0 && y<=win_heidht)
    video_buffer[x+y*640] = color;

} // if
```

Теперь, когда вы поняли, в чем суть отсечения, и научились его реализовать, перейдем к отсечению части битового образа.

Трудоемкий способ отсечения битовых образов

Отсечение битовых образов не сложнее отсечения пикселей. Его можно осуществить двумя способами.

- **Метод 1.** Независимое отсечение каждого пикселя, входящего в состав битового образа, по мере генерирования этих пикселей. Этот метод простой, но медленный.
- **Метод 2.** Отсечение части ограничивающего прямоугольника, которая выходит за рамки области видимости, с последующим отображением на экране только той части изображения, которая находится в области видимости. Этот метод более сложный, но зато работает очень быстро и почти не приводит к потере производительности.

Очевидно, мы прибегнем ко второму способу, принцип работы которого проиллюстрирован на рис. 7.18. Предположим, что экран занимает область от (0,0) до (SCREEN_WIDTH-1,SCREEN_HEIGHT-1), верхний левый угол битового образа расположен в точке с координатами (x,y), а его правый нижний угол — в точке с координатами (x+width-1, y+height-1). Прервитесь на минуту и убедитесь, что вы понимаете, зачем от некоторых координат отнимается единица. По сути, если исходный битовый образ имеет размеры 1×1, то его ширина и высота равны одному пикселю. Поэтому, если одна из угловых точек битового образа имеет координаты (x,y), то координаты его второй угловой точки — (x+1-1,y+1-1).

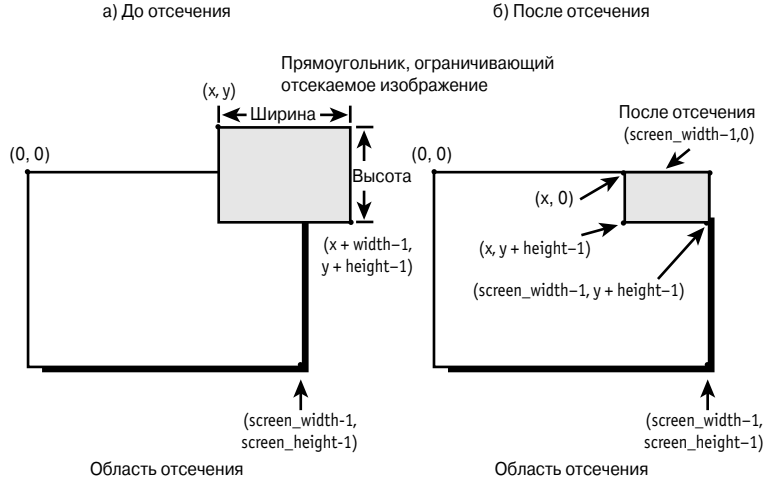


Рис. 7.18. Отсечение ограничивающего прямоугольника

План кампании по осуществлению отсечения прост — достаточно определить прямоугольник, который находится в области видимости, а затем вывести на экран только эту часть битового образа. Ниже приведен код, который выполняет описанное выше действие в линейном режиме 640×480×8.

```
// Размеры окна или области видимости с началом в точке (0,0)
#define SCREEN_WIDTH  640
#define SCREEN_HEIGHT 480
```

```
void Blit_Clipped(int x, int y,  // Начало битового образа
int width, int height,           // Размер битового образа в пикселях
UCHAR *bitmap,                   // Данные битового образа
UCHAR *video_buffer,             // Указатель на поверхность
                                  // в видеобуфере
int   mempitch)                   // Шаг видеопамяти
{
// Эта функция предназначена для блиттинга и отсечения
// поверхности назначения с указателем video_buffer.
// Предполагается, что установлен режим 640x480x8.
// Эта функция несколько отличается от описанной
// в книге — в ней не предполагается линейность шага

// Сначала реализуем тривиальный случай: видима ли
// хоть какая-то часть битового образа?
if ((x >= SCREEN_WIDTH) || (y>= SCREEN_HEIGHT) ||
   ((x + width) <= 0) || ((y + height) <= 0))
   return;

// Отсечение исходного прямоугольника;
// предварительное вычисление границ прямоугольника
int x1 = x;
int y1 = y;
int x2 = x1 + width — 1;
int y2 = y1 + height -1;

// Обработка верхнего левого угла
if (x1 < 0)
  x1 = 0;

if (y1 < 0)
  y1 = 0;

// Обработка нижнего правого угла
if (x2 >= SCREEN_WIDTH)
   x2 = SCREEN_WIDTH-1;

if (y2 >= SCREEN_HEIGHT)
   y2 = SCREEN_HEIGHT-1;

// Теперь известно, что выводить нужно только ту
// часть битового образа, которая расположена между
// точками (x1,y1) и (x2,y2);
// определяем сдвиги битового образа по осям x и y,
// чтобы вычислить начальную точку растеризации
int x_off = x1 — x;
int y_off = y1 — y;

// Вычисление количества столбцов и строк,
// участвующих в блиттинге
int dx = x2 — x1 + 1;
int dy = y2 — y1 + 1;
```

```
// Вычисление начального адреса в видеобуфере
video_buffer += (x1 + y1*mempitch);

// Вычисление в битовом образе начального адреса
// для сканирования данных
bitmap += (x_off + y_off*width);

// Теперь переменная bitmap указывает на первый
// пиксель битового образа, который нужно переслать,
// а переменная video_buffer — на ячейку памяти в
// буфере назначения; организуем цикл растеризации

UCHAR pixel; // Применяется для считывания и записи пикселей

for (int index_y = 0; index_y < dy; index_y++)
  {
  // Внутренний цикл, в котором происходит действие
  for (int index_x = 0; index_x < dx; index_x++)
    {
    // Считывание пикселя из исходного битового образа;
    // проверка на прозрачность и вывод
    if ((pixel = bitmap[index_x]))
      video_buffer[index_x] = pixel;

    } // for index_x

    // Сдвиг указателей
    video_buffer+=mempitch; // Количество байтов в
                            // строке сканирования
    bitmap     +=width;     // Количество байтов в
                            // строке битового образа

  } // for index_y

} // Blit_Clipped
```

Чтобы продемонстрировать работу этого небольшого программного отсекателя, был создан простейший пример обработки битовых образов. Он представляет собой массив, состоящий из 64 байтов, в котором содержится изображение маленькой улыбающейся физиономии. Вот объявление этого массива:

```
UCHAR happy_bitmap[64] = {0,0,0,0,0,0,0,0,
                          0,0,1,1,1,1,0,0,
                          0,1,0,1,1,0,1,0,
                          0,1,1,1,1,1,1,0,
                          0,1,0,1,1,0,1,0,
                          0,1,1,0,0,1,1,0,
                          0,0,1,1,1,1,0,0,
                          0,0,0,0,0,0,0,0};
```

Затем система переводится в режим заднего буфера 320×240×8 и создается цветовой индекс RGB (255,255,0), соответствующий желтому цвету. После этого улыбающаяся физиономия перемещается по экрану с постоянной скоростью, величина которой выбрана случайным образом. Когда изображение приближается слишком близко к краю экрана, вступает в игру

функция-отсекатель. Я, пожалуй, немного увлекся и сделал целых 100 улыбающихся физиономий! Конечный вариант программы содержится в файле DEMO7_8.CPP; на рис. 7.19 приведена копия экрана, на который выводится результат работы программы.

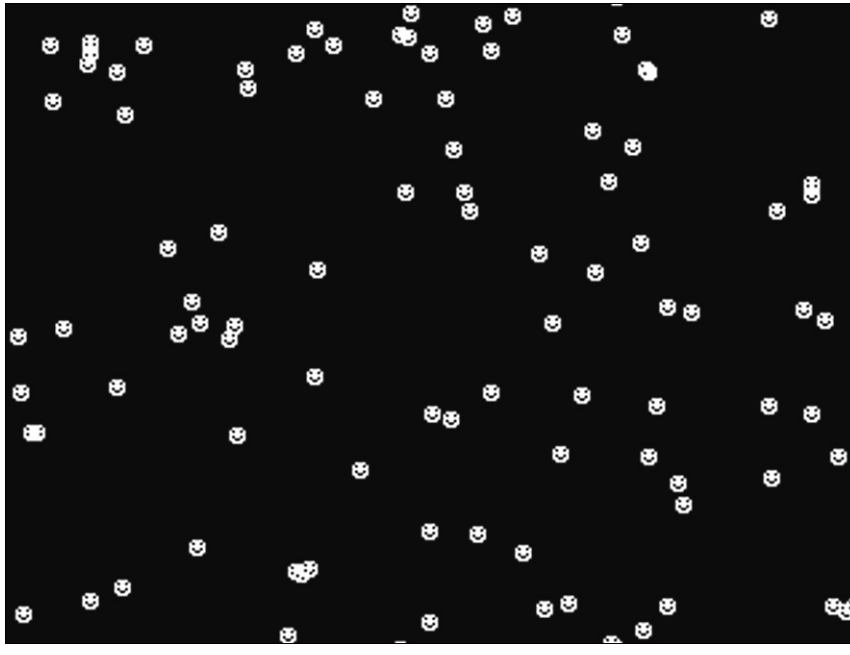


Рис. 7.19. Программа DEMO7_8.EXE в действии

Ниже приведен листинг функции Game_Main(), которая входит в состав рассматриваемой программы.

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
// Главный цикл игры, в котором выполняется вся ее логика

DDBLTFX ddbltfx; // fx-структура блиттера

// Подстраховка против повторного запуска
if (window_closed)
   return(0);

// Проверка, не нажал ли пользователь клавишу <Esc>
// для выхода из игры
if (KEYDOWN(VK_ESCAPE))
   {
   PostMessage(main_window_handle,WM_CLOSE,0,0);
   window_closed = 1;
   } // if

// Очистка заднего буфера с помощью блиттера;
// сначала инициализируем структуру DDBLTFX
DDRAW_INIT_STRUCT(ddbltfx);
```

```
// Задаем нужный цвет
ddbltfx.dwFillColor = 0;

// Вызов блиттера
if (FAILED(lpddsback->Blt(
     NULL, // Указатель на область назначения
     NULL, // Указатель на исходную поверхность
     NULL, // Указатель на исходную область
     DDBLT_COLORFILL | DDBLT_WAIT,
         // Заполнение цветом и ожидание
     &ddbltfx)))
         // Указатель на структуру DDBLTFX
         // с необходимой информацией
return(0);

// Инициализация ddsd
DDRAW_INIT_STRUCT(ddsd);

// Блокировка поверхности в заднем буфере
if (FAILED(lpddsback->Lock(NULL,&ddsd,
        DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR,
        NULL)))
   return(0);


// Вывод всех физиономий
for (int face=0; face < 100; face++)
   {
   Blit_Clipped(happy_faces[face].x,
            happy_faces[face].y,
            8,8,
            happy_bitmap,
            (UCHAR *)ddsd.lpSurface,
            ddsd.lPitch);
   } // for face

// Перемещение физиономий
for (face=0; face < 100; face++)
   {
   // Перемещение
   happy_faces[face].x+=happy_faces[face].xv;
   happy_faces[face].y+=happy_faces[face].yv;

   // Если изображения выходят за экран,
   // возвращаем их с другой стороны
   if (happy_faces[face].x > SCREEN_WIDTH)
      happy_faces[face].x = -8;
   else
   if (happy_faces[face].x < -8)
      happy_faces[face].x = SCREEN_WIDTH;

   if (happy_faces[face].y > SCREEN_HEIGHT)
```

```
      happy_faces[face].y = -8;
    else
    if (happy_faces[face].y < -8)
      happy_faces[face].y = SCREEN_HEIGHT;

    } // face

// Снятие блокировки с поверхности
if (FAILED(lpddsback->Unlock(NULL)))
  return(0);

// Вывод страницы
while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));

// Небольшая пауза
Sleep(30);

// Успешное завершение (или ваш код возврата)
return(1);

} // Game_Main
```

НА ЗАМЕТКУ

Внимательно просмотрите код функции `Blit_Clipped()` в демонстрационной программе, потому что он слегка изменен. Этот код приспособлен для работы с переменными, в которых может храниться шаг памяти. В этом нет ничего особенного, но у некоторых читателей могут возникнуть вопросы. Кроме того, вы можете поинтересоваться, почему это автор решил воспользоваться режимом 320x240. Просто оказалось, что небольшой битовый образ размерами 8x8 в режиме 640x480 выглядит таким маленьким, что можно ослепнуть, пытаясь его разглядеть.

Отсечение с помощью интерфейса IDirectDrawClipper

Пришло время сравнить, насколько легче выполнять отсечение в DirectDraw по сравнению с отсечением с помощью программ собственного производства (о том, сколько работы для этого нужно выполнить, вы получили представление в предыдущем подразделе). В состав DirectDraw входит интерфейс под названием `IDirectDrawClipper`, с помощью которого осуществляется блиттинг и отсечение двухмерных изображений, а также 3D-растеризация под Direct3D. Разумеется, с этой темой стоило бы ознакомиться получше, но в данный момент нас интересует лишь применение этого интерфейса для отсечения битовых образов, которые подвергаются блиттингу с помощью функции `Blt()`, а также использование соответствующего аппаратного блиттера.

Чтобы использовать отсечение в DirectDraw, выполните такие действия.

1. Создайте объект-отсекатель DirectDraw.
2. Составьте список объектов, подвергаемых отсечению.
3. Передайте этот список отсекателю с помощью функции `IDIRECTDRAWCLIPPER::SetClipList()`.
4. Присоедините отсекатель к окну и (или) поверхности с помощью функции `IDIRECTDRAWSURFACE7::SetClipper()`.

Приступим к выполнению первого этапа. Функция, предназначенная для создания интерфейса `IDirectDrawClipper`, носит название `IDIRECTDRAWSURFACE7::CreateClipper()`; ниже приведен ее прототип.

```
HRESULT CreateClipper(DWORD dwFlags,          // Управляющие флаги
  LPDIRECTDRAWCLIPPER FAR *lplpDDClipper, // Адрес указателя
                                          // на интерфейс
  IUnknown FAR        *pUnkOuter);        // У нас - NULL
```

В случае успешного завершения эта функция возвращает значение DD_OK.

Смысл параметров функции CreateClipper() понять несложно. Параметр dwFlags в данном случае не используется, поэтому его значение должно быть равным 0. Параметр lplpDDClipper — это адрес интерфейса IDirectDrawClipper; после успешного завершения работы функции он указывает на отсекатель DirectDraw. О параметре pUnkOuter не стоит беспокоиться — просто задайте для него значение NULL. Чтобы создать объект-отсекатель, достаточно вставить в программу приведенный ниже код.

```
LPDIRECTDRAWCLIPPER lpddclipper = NULL; // Для хранения отсекателя
if (FAILED(lpdd->CreateClipper(0,&lpddclipper,NULL)))
  return(0);
```

Если функция выполняется успешно, указатель lpddclipper будет указывать на достоверный интерфейс IDirectDrawClipper, и с помощью этого указателя можно будет вызывать различные методы.

Все это хорошо, но как создать список предназначенных для отсечения объектов и что этот список будет собой представлять? Как видно из рис. 7.20, список отсечения DirectDraw — это список хранящихся в структурах RECT прямоугольников, указывающих на достоверные области, в которые можно осуществлять блиттинг. Из рисунка видно, что на поверхность дисплея выводится несколько прямоугольников и система блиттинга DirectDraw может выполнять пересылку только в эти прямоугольные области. Изображение можно выводить в любое место экрана, но аппаратный блиттер способен выводить его *только* в областях отсечения, которые получили название *список отсечения* (clipping list).

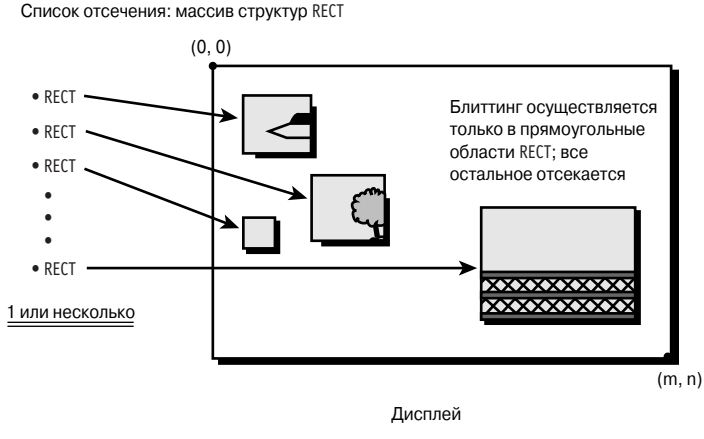


Рис. 7.20. Взаимосвязь между списком отсечения и блиттером

Чтобы создать список отсечения, необходимо заполнить довольно неприятную структуру под названием RGNDATA (Region Data — данные областей), которая приведена ниже.

```
typedef struct _RGNDATA
{
  RGNDATAHEADER rdh; // Заголовочная информация
  char Buffer[1];          // Список структур RECT
} RGNDATA;
```

Эта структура данных выглядит немного странно. Она представляет собой структуру переменной длины, т.е. входящий в состав структуры массив Buffer[] может быть произвольной длины. Рассматриваемая структура генерируется не статически, а динамически, и ее истинная длина хранится в переменной типа RGNDATAHEADER. Здесь представлена старая версия новой методики построения структур данных в DirectDraw, согласно которой каждая структура содержит поле dwSize, где задается ее размер. Возможно, лучше было бы в качестве члена структуры использовать не массив Buffer[], предназначенный для хранения битовых значений, а указатель на этот массив.

Все, что необходимо сделать, — это выделить достаточное количество памяти для хранения структуры RGNDATAHEADER, а также для хранения массива, состоящего из одного или нескольких структур RECT, которые размещаются в смежных областях памяти (рис. 7.21). Затем останется только выполнить преобразование к типу RGNDATA и осуществить передачу данных.

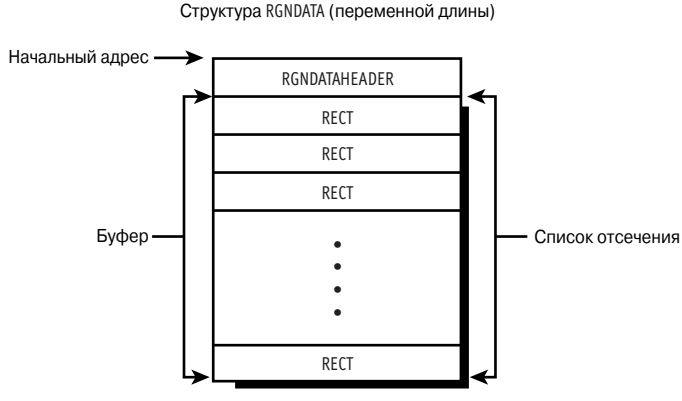


Рис. 7.21. Схема размещения в памяти элементов структуры RGNDATA

Рассмотрим содержимое структуры RGNDATAHEADER.

```
typedef struct _RGNDATAHEADER
{
  DWORD dwSize;       // Размер заголовка в байтах
  DWORD iType;        // Тип данных, описывающих область
  DWORD nCount;       // Количество элементов массива Buffer[]
  DWORD nRgnSize;     // Размер массива Buffer[]
  RECT rcBound;       // Прямоугольник, окаймляющий все области
} RGNDATAHEADER;
```

Чтобы задать эту структуру, присвоим переменной dwSize значение sizeof(RGNDATAHEADER), переменной iType — значение RDH_RECTANGLES, переменной nCount — количество входящих в список областей RECT, переменной nRgnSize — выраженный в байтах размер массива Buffer[] (который равен произведению sizeof(RECT)*nCount). Кроме того, нужно создать прямоугольник, ограничивающий все области типа RECT, и сохранить его в переменной rcBound. После создания структуры RGNDATA ее нужно передать отсекателю с помощью функции IDIRECTDRAWCLIPPER::SetClipList(), прототип которой приведен ниже.

```
HRESULT SetClipList(
  LPRGNDATA lpClipList,  // Указатель на структуру RGNDATA
  DWORD     dwFlags);    // Флаги; всегда используем 0
```

Здесь, собственно, не о чем говорить. Если предположить, что предназначенная для списка отсечения структура RGNDATA уже сгенерирована, то сам список задается так:

```
if (FAILED(lpddclipper->SetClipList(&rgndata,0)))
  return(0);
```

После того как список отсечения задан, появляется возможность присоединить отсекатель к поверхности с помощью функции IDIRECTDRAWSURFACE7::SetClipper(). Это можно сделать, воспользовавшись приведенной ниже строкой.

```
HRESULT SetClipper(LPDIRECTDRAWCLIPPER lpDDClipper);
```

Вот как выглядит эта функция в действии:

```
if (FAILED(lpddsurface->SetClipper(&lpddclipper)))
  return(0);
```

В большинстве случаев lpddsurface указывает на внеэкранные поверхности, например на поверхности в заднем буфере. (Обычно к первичной поверхности отсекатель не присоединяется.)

Вероятно, вам уже надоело вникать во все эти подробности, касающиеся создания структуры RGNDATA и ее настройки. Причина такого стиля изложения в том, что слишком сложно раскрывать эту тему последовательно, шаг за шагом; легче просто посмотреть на код. Именно с этой целью создана функция под названием DDraw_Attach_Clipper() (она входит в графическую библиотеку T3D), в которой создается отсекатель и список отсечения, а затем все это присоединяется к поверхности. Ниже приведен код этой функции.

```
LPDIRECTDRAWCLIPPER DDraw_Attach_Clipper
            (LPDIRECTDRAWSURFACE7 lpdds,
             int num_rects,
             LPRECT clip_list)

{
// Эта функция создает отсекатель из переданного ей списка
// отсечения и присоединяет этот список к поверхности,
// заданной в списке аргументов

int index;                          // Переменная цикла
LPDIRECTDRAWCLIPPER lpddclipper;  // Указатель на создаваемый отсекатель
LPRGNDATA region_data;              // Указатель на область данных, в которой
                                    // содержатся заголовок и список отсечения

// Создание отсекателя DirectDraw
if (FAILED(lpdd->CreateClipper(0,&lpddclipper,NULL)))
  return(NULL);

// Пользуясь полученными функцией данными, создаем список
// отсечения

// Выделение памяти для области данных
region_data = (LPRGNDATA)malloc(sizeof(RGNDATAHEADER)+
        num_rrects*sizeof(RECT));

// Копирование прямоугольников в область данных
memcpy(region_data->Buffer, clip_list,
    sizeof(RECT)*num_rects);
```

```
// Задаем поля заголовка
region_data->rdh.dwSize  = sizeof(RGNDATAHEADER);
region_data->rdh.iType   = RDH_RECTANGLES;
region_data->rdh.nCount  = num_rects;
region_data->rdh.nRgnSize = num_rects*sizeof(RECT);
region_data->rdh.rcBound.left  =  64000;
region_data->rdh.rcBound.top   =  64000;
region_data->rdh.rcBound.right  = -64000;
region_data->rdh.rcBound.bottom = -64000;

// Нахождение границ всех областей отсечения
for (index=0; index<num_rects; index++)
{
   // Проверка очередного прямоугольника: не
   // выходит ли он за рамки текущей границы
   if (clip_list[index].left <
      region_data->rdh.rcBound.left)
      region_data->rdh.rcBound.left =
         clip_list[index].left;

   if (clip_list[index].right >
      region_data->rdh.rcBound.right)
      region_data->rdh.rcBound.right =
         clip_list[index].right;

   if (clip_list[index].top <
      region_data->rdh.rcBound.top)
      region_data->rdh.rcBound.top =
         clip_list[index].top;

   if (clip_list[index].bottom >
      region_data->rdh.rcBound.bottom)
      region_data->rdh.rcBound.bottom =
         clip_list[index].bottom;

} // for index

// Задаем список отсечения
if (FAILED(lpddclipper->SetClipList(region_data, 0)))
{
   // Очистка памяти и возврат кода ошибки
   free(region_data);
   return(NULL);
} // if

// Присоединение отсекателя к поверхности
if (FAILED(lpdds->SetClipper(lpddclipper)))
{
   // Очистка памяти и возврат кода ошибки
   free(region_data);
   return(NULL);
} // if
```

```
// Успешное завершение; освобождение памяти и
// возврат указателя на новый отсекатель
free(region_data);
return(lpddclipper);

} // DDraw_Attach_Clipper
```

Эта функция очень проста в применении. Предположим, у нас есть система анимации, в которую входят первичная поверхность с именем lpddsprimary и вторичный задний буфер с именем lpddsback, к которому нужно присоединить отсекатель с таким списком отсечения:

```
RECT rect_list[3] = {{10,10,50,50},
                     {100,100,200,200}
                     {300,300,500,450}};
```

Вот как выглядит в этом случае вызов функции:

```
LPDIRECTDRAWCLIPPER lpddclipper =
      DDraw_Attach_Clipper(lpddsback,3,rect_list);
```

Здорово, правда? Если воспользоваться таким вызовом, на экране будут отображаться только те части битовых образов, которые находятся в прямоугольниках с угловыми точками (10, 10) и (50, 50), (100, 100) и (200, 200), (300, 300) и (500, 450). Эта функция входит в библиотеку, созданную мною в процессе написания этой главы. Позже я опишу все функции, из которых состоит эта библиотека, и тогда отпадет необходимость писать весь этот скучный код DirectDraw самому и можно будет сосредоточить внимание на программировании самой игры.

С помощью приведенного выше кода создана демонстрационная программа под названием DEMO7_9.CPP. В ее основу положена программа DEMO7_7.CPP, демонстрирующая блиттинг. В программе DEMO7_9.CPP осуществлен переход к 8-битовым цветам и добавлена функция отсечения, чтобы блиттинг осуществлялся в отображаемые на первичной поверхности области отсечения. Кроме того, для соблюдения последовательности изложения выбраны те области отсечения, о которых только что шла речь. На рис. 7.22 показана копия экрана, на который выводится результат работы программы DEMO7_9.EXE. Обратите внимание, что экран выглядит как набор небольших окон, в которых отсекатель разрешает отображение.

Ниже приведен код программы DEMO7_9.CPP, входящий в функцию Game_Main(), в котором задаются параметры областей отсечения.

```
// Создание и присоединение отсекателя
RECT rect_list[3] = {{10,10,50,50},
                     {100,100,200,200}
                     {300,300,500,450}};

if (FAILED(lpddclipper =
   DDraw_Attach_Clpper(lpddsprimary,3,rect_list)))
     return(0);
```

Конечно же, не имеет значения, в каком режиме происходит добавление отсекателя — в 16-битовом или более высоком; форма вызова функции от этого не зависит. На работу отсекателя режим визуального отображения не влияет, поскольку отсечение происходит на другом, более абстрактном уровне, а потому количество приходящихся на пиксель битов не имеет значения.

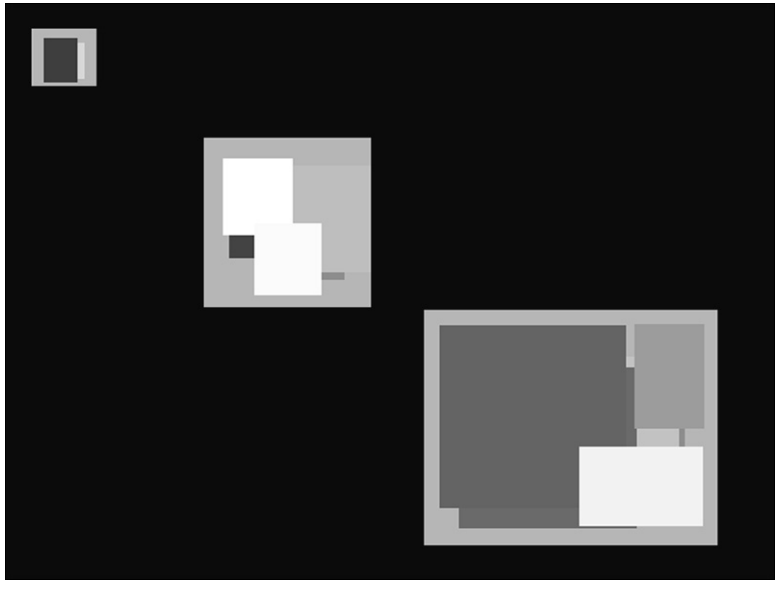


Рис. 7.22. Программа DEMO7_9.EXE в действии

Отлично! А теперь хватит возиться с градиентными заливками и цветными прямоугольниками. Все это уже смертельно наскучило. Пора переходить к нормальным битовым образам. В следующем разделе будет показано, как в системе Windows загружать битовые образы.

Работа с битовыми образами

Существует огромное множество различных растровых форматов, однако при создании игр чаще всего применяется несколько: .PCX (формат программы Paint), .TGA (Targa — усовершенствованный адаптер растровой графики) и .BMP (стандартный формат системы Windows). Все перечисленные форматы имеют свои за и против, однако, поскольку используется операционная система Windows, во избежание непредвиденных затруднений лучше всего применять "родной" формат этой системы .BMP. Конечно же, по этому поводу могут быть различные мнения.

При работе с другими форматами все действия выполняются аналогично. Поэтому, после того как научишься применять один формат, освоение других сводится к умению загрузить заголовочную структуру формата и считать с диска некоторые байты. Например, многие начинают с формата .PNG (Portable Network Graphics — формат переносимой сетевой графики).

Загрузка файлов в формате .BMP

Файлы в формате .BMP можно считать несколькими способами. Можно самому написать специальную программу для считывания, воспользоваться функцией Win32 API или применить комбинацию этих двух подходов. Поскольку постичь принцип действия функций Win32 обычно так же сложно, как и написать свои собственные, можно с равным успехом самостоятельно написать загрузчик формата .BMP. Файл в этом формате состоит из трех частей (рис. 7.23).

Заголовок .BMP-файла. В заголовке файлов в этом формате содержится общая информация о битовом образе; эта информация представлена структурой данных BITMAPFILEHEADER, входящей в состав Win32.

```
typedef struct tagBITMAPFILEHEADER
{ // заголовок файла в формате .BMP
  WORD  bfType;        // Определяет тип файла
                       // Для .bmp должен быть равен 0x4D42
  DWORD bfSize;        // Задает размер файла в байтах
  WORD  bfReserved1;   // Зарезервированный параметр;
                       // должен быть равен 0
  WORD  bfReserved2;   // Зарезервированный параметр;
                       // должен быть равен 0
  DWORD bfOffBits;     // Задает выраженное в байтах
                       // смещение входящих в образ
                       // битов относительно структуры
                       // BITMAPFILEHEADER
} BITMAPFILEHEADER;
```

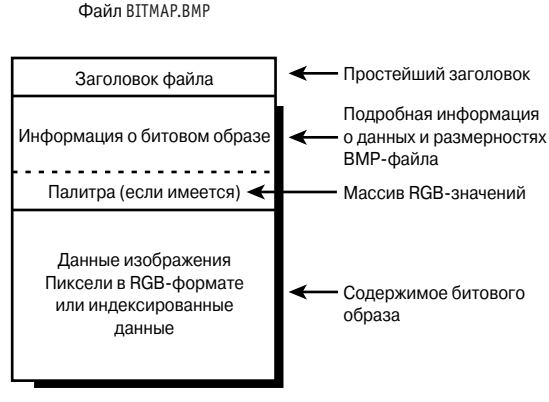


Рис. 7.23. Структура .BMP-файла

Раздел с информацией о битовом образе. Эта часть состоит из двух структур данных — BITMAPINFOHEADER и информации о палитре (если таковая имеется в наличии).

```
typedef struct tagBITMAPINFO
{ // Информация о битовом образе
  BITMAPINFOHEADER bmiHeader;  // Заголовочная информация
  RGBQUAD      bmiColors[1];   // Палитра (если есть)
} BITMAPINFO;
```

Ниже приведена структура BITMAPINFOHEADER.

```
typedef struct tagBITMAPINFOHEADER{ // Информация о битовом образе
  DWORD biSize;   // Количество требующихся для структуры байтов
  LONG biWidth;   // Ширина битового образа в пикселях
  LONG biHeight; // Высота битового образа в пикселях
                  // Если biHeight>0, битовый образ
                  // строится снизу вверх, и его начало
                  // находится в нижнем левом углу; если
                  // biHeight<0, битовый образ строится
                  // сверху вниз и его начало находится
                  // в верхнем левом углу;
  WORD biPlanes; // Количество цветовых плоскостей;
                  // должно быть равно 1
```

```
  WORD biBitCount;        // Количество битов в пикселе; этот
                          // параметр должен быть равен 1, 4, 8, 16, 24 или 32
  DWORD biCompression; // Тип сжатия (параметр для опытных
                          // программистов); для несжатых
                          // .bmp-файлов, которые мы
                          // собираемся использовать, этот
                          // параметр равен значению BI_RGB
  DWORD biSizeImage;    // Размер изображения в байтах
  LONG biXPelsPerMeter; // Количество пикселей в метрепо оси X
  LONG biYPelsPerMeter; // Количество пикселей в метре по оси Y
  DWORD biClrUsed;        // Количество цветов в изображении
  DWORD biClrImportant; // Количество важных цветов
} BITMAPINFOHEADER;
```

НА ЗАМЕТКУ

Для 8-битовых изображений в полях biClrUsed и biClrIpmortant обычно задается значение 256, а для 16- и 24-битовых — значение 0. Таким образом, всегда следует проверять значение поля biBitCount, чтобы узнать, сколько битов включает каждый пиксель, и далее исходить из этих данных.

Область данных битового образа. В этой части файла задается байтовый поток (в сжатой или обычной форме), описывающий изображение в 1-, 4-, 8-, 16- или 24-битовом формате. Данные располагаются последовательно, строка за строкой, однако они могут следовать и в обратном порядке, когда первая строка данных соответствует последней строке изображения (рис. 7.24). Порядок следования данных можно узнать по знаку поля biHeight: положительное значение этого поля свидетельствует о том, что битовый образ строится вверх ногами, а отрицательное — о том, что он строится нормально.

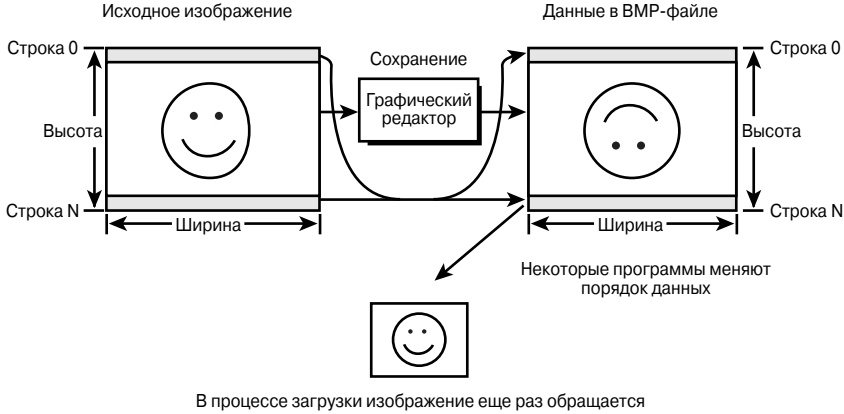


Рис. 7.24. Иногда данные, описывающие изображение, располагаются в .bmp-файле в обратном порядке по оси y

Чтобы вручную считать файл в формате .bmp, его сначала нужно открыть (любым способом ввода-вывода файлов) и считать все данные, которые входят в структуру BITMAPFILEHEADER. Затем следует считать раздел BITMAPINFO, представляющий собой структуру BITMAPINFOHEADER с добавленной к ней палитрой (если это 256-цветовое изображение). После этого нужно определить размер битового образа (biWidth, biHeight), а также глубину цвета (biBitCount, biClrUsed). Кроме того, потребуется считать данные, описывающие сам битовый образ, а также палитру (если она есть). Конечно же, в изложенном

выше процессе есть много таких дополнительных деталей, как выделение буферов для считываемых данных и манипуляция с указателями файлов. Кроме того, каждая входящая в палитру запись представляет собой структуру RGBQUAD, в которой обычные значения PALETEENTRY следуют в строго определенном порядке; поэтому данные записи нужно преобразовать следующим образом:

```
typedef struct tagRGBQUAD
{ // rgbq
  BYTE rgbBlue;        // синий
  BYTE rgbGreen;       // зеленый
  BYTE rgbRed;         // красный
  BYTE rgbReserved;    // не используется
} RGBQUAD;
```

Возможно, вы еще помните функцию LoadBitmap(), которая использовалась в главе 4, "GDI, управляющие элементы и прочее", для загрузки с диска растровых ресурсов. Можно пользоваться этой функцией, но тогда при компиляции исполняемых файлов игры всегда следует задавать все битовые образы игры как ресурсы. Хотя в целом это неплохо, на этапе разработки иногда лучше пойти другим путем. Желательно иметь возможность доработать графику с помощью Paint или другой программы, сбросить полученный битовый образ в каталог, а затем запустить игру и посмотреть, как повлияли изменения на работу программы. Таким образом, требуется более общая функция, считывающая файлы с битовыми образами, к разработке которой мы и приступим через некоторое время. А перед этим рассмотрим функцию под названием LoadImage(), входящую в программный интерфейс приложения Win32 и предназначенную для загрузки битовых образов.

```
HANDLE LoadImage(
 HINSTANCE hint,     // Указатель на экземпляр,
                     // содержащий изображение
 LPCTSTR lpszName,   // Название или идентификатор
                     // изображения
 UNIT   uType,       // Тип изображения
 int    cxDesired,   // Требуемая ширина
 int    cyDesired,   // Требуемая высота
 UNIT   fuLoad);     // Флаги загрузки
```

Эта функция носит довольно общий характер, однако пока что нам она нужна только для загрузки файлов с диска, поэтому не будем обращать внимания на другие ее возможности. Чтобы выполнить с помощью этой функции нужное действие, достаточно задать значения ее параметров так, как показано ниже.

hinst — указатель на экземпляр. Зададим его значение равным NULL.

lpszName — имя .bmp-файла на диске. Задаем обычное имя файла, например ANDRE.BMP, C:\images\ship.bmp и т.д.

uType — тип загружаемого изображения. Задаем значение IMAGE_BITMAP.

cxDesired, cyDesired — требующиеся ширина и высота битового образа. Если задать в качестве этих значений любые отличные от нуля числа, функция LoadImage() изменит масштаб битового образа в соответствии с ними. Таким образом, если размер изображения известен, его следует задать. В противном случае значения этих параметров задаются равными нулю, а размер изображения считывается позже.

fuLoad — управляющие флаги загрузки. Задаем значение (LR_LOADFROMFILE | LR_CREATEDIBSECTION). Указывает функции LoadImage(), что данные с диска следует загружать с помощью имени, хранящегося в переменной lpszName, и что не следует преобразовывать данные битового образа в соответствии с цветовыми характеристиками дисплея.

Единственная проблема, возникающая при работе с функцией LoadImage(), заключается в том, что она слишком обобщенная, и извлечь с ее помощью данные не так просто. Чтобы получить доступ к заголовочной информации, потребуются дополнительные функции, а если имеется палитра, то возникают специфические трудности. Вместо этого я разработал функцию, позволяющую загружать с диска битовые образы в любом формате (включая форматы с палитрой) и заполнять информацией такую структуру:

```
typedef struct BITMAP_FILE_TAG
{
  BITMAPFILEHEADER bitmapfileheader;     // Здесь содержится
                                         // заголовок растрового файла
  BITMAPINFOHEADER bitmapinfoheader; // Здесь содержатся
                                      // все данные, включая палитру
  PALETTEENTRY palette[256];         // Здесь хранится палитра
  UCHAR *buffer;                     // Указатель на данные

} BITMAP_FILE, *BITMAP_FILE_PTR;
```

Обратите внимание, что структуры BITMAPINFOHEADER и BITMAPINFO, по сути, объединены в одну. Так с ними намного проще работать. Теперь приведем код функции Load_Bitmap_File().

```
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, char *filename)
{
// Функция открывает растровый файл и загружает данные
// в битовый образ

int file_handle,  // Индекс файла
  index;          // Индекс цикла

UCHAR  *temp_buffer = NULL; // Преобразование 24-битовых
                            // изображений в 16-битовые
OFSTRUCT file_data;         // Информация о содержащихся
                            // в файле данных

// Открытие файла, если он существует
if ((file_handle = OpenFile(filename,&file_data,
                  OF_READ))==-1)
  return(0);

// Загрузка заголовка растрового файла
_lread(file_handle, &bitmap->bitmapfileheader,
     sizeof(BITMAPFILEHEADER));

// Проверка типа файла
if (bitmap->bitmapfileheader.bfType!=BITMAP_ID)
{
  // Закрытие файла
  _lclose(file_handle);

  // Ошибка
  return(0);
} // if
```

```
// Установлено, что файл растровый;
// считывание остальных разделов

// Считывание заголовочной информации

// Загрузка заголовка растрового файла
_lread(file_handle, &bitmap->bitmapinfoheader,
      sizeof(BITMAPINFOHEADER));

// Загрузка цветовой палитры при ее наличии
if (bitmap->bitmapinfoheader.biBitCount == 8)
{
   _lread(file_handle, &bitmap->palette,
      MAX_COLORS_PALETTE*sizeof(PALETTEENTRY));

   // Установка всех флагов палитры и установка обратного
   // BGR-формата данных структуры RGBQUAD
   for (index=0; index < MAX_COLORS_PALETTE; index++)
   {
      // меняем местами поля, соответствующие красному
      // и синему цветам
      int temp_color = bitmap->palette[index].peRed;
      bitmap->palette[index].peRed  =
               bitmap->palette[index].peBlue;
      bitmap->palette[index].peBlue = temp_color;

      // Всегда задавайте флаги
      bitmap->palette[index].peFlags = PC_NOCOLLAPSE;
   } // for index

} // if

// Загрузка данных, описывающих изображение
_lseek(file_handle,
     -(int)(bitmap->bitmapinfoheader.biSizeImage),
     SEEK_END);

// Считываем изображение;

if (bitmap->bitmapinfoheader.biBitCount==8 ||
   bitmap->bitmapinfoheader.biBitCount==16 ||
   bitmap->bitmapinfoheader.biBitCount==24)
{
   // Удаление прошлого изображения (если оно есть)
   if (bitmap->buffer)
      free(bitmap->buffer);

   // Выделение памяти для изображения
   if (!(bitmap->buffer =
     (UCHAR *)malloc(bitmap->bitmapinfoheader.biSizeImage)))
   {
      // Закрытие файла
      _lclose(file_handle);
```

```
      // Ошибка
      return(0);
    } // if

   // Считывание
    _lread(file_handle,bitmap->buffer,
         bitmap->bitmapinfoheader.biSizeImage);

} // if
else
{
   // Серьезная проблема
   return(0);

} // else

// Закрытие файла
_lclose(file_handle);

// Отображение битового образа
Flip_Bitmap(bitmap->buffer,
         bitmap->bitmapinfoheader.biWidth*
        (bitmap->bitmapinfoheader.biBitCount/8),
         bitmap->bitmapinfoheader.biHeight);

// Успешное выполнение
return(1);

} // Load_Bitmap_File
```

На самом деле эта функция не такая большая и сложная, как кажется; просто ее долго писать, вот и все.

НА ЗАМЕТКУ

В заключительной части приведенной функции вызывается функция Flip_Bitmap(), которая просто обращает изображение. Дело в том, что строки в большинстве файлов .bmp расположены в обратном порядке. Функция Flip_Bitmap() входит в разработанную автором библиотеку; кроме того, она скопирована в демонстрационный пример, который будет приведен позже, поэтому у вас еще будет возможность ознакомиться с этой функцией.

Функция Load_Bitmap_File() открывает растровый файл, загружает заголовки, а затем — изображение и палитру (если изображение представляет собой 256-цветный битовый образ). Функция работает с изображениями, созданными в 8-, 16- и 24-битовых режимах. Однако, независимо от формата изображения, буфер, содержащий буфер изображения, — это просто указатель на байт, поэтому над ним необходимо выполнить преобразование типа или некоторые арифметические операции (если это 16- или 24-битовое изображение). Кроме того, в функции выделяется динамический буфер для изображения, который после выполнения всех действий с изображением необходимо освободить. Это можно выполнить с помощью приведенной ниже функции Unload_Bitmap_File().

```
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap)
{
// Функция освобождает память, выделенную для изображения
if (bitmap->buffer)
```

```
{
  // Освобождение памяти
  free(bitmap->buffer);

  // Сброс указателя
  bitmap->buffer = NULL;

} // if

// Успешное завершение
return(1);

} // Unload_Bitmap_File
```

Вскоре вы научитесь загружать в память растровые файлы и отображать на экране их содержимое, но сначала рассмотрим, что можно делать с изображениями в общем контексте игры.

Использование битовых образов

Большинство игр предоставляют разнообразные возможности для творчества художников. В играх встречаются двухмерные спрайты, двухмерные текстуры, трехмерные модели и т.д. В большинстве случаев двухмерные картинки загружаются последовательно, кадр за кадром, в виде отдельных изображений (рис. 7.25) или в виде шаблонов, т.е. как несколько однотипных изображений, из которых составлена прямоугольная матрица (рис. 7.26). Оба метода имеют свои за и против. Преимущество загрузки отдельных изображений, каждое из которых хранится в своем файле, заключается в том, что данные из файла при смене изображений можно использовать непосредственно. Однако в игре могут фигурировать сотни кадров анимации, из которых формируется персонаж двухмерной игры. Это привело бы к необходимости обрабатывать сотни тысяч отдельных файлов .bmp!

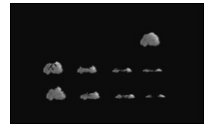


Рис. 7.25. Стандартный набор битовых образов без шаблона

Замечательная особенность шаблонных изображений, подобных показанным на рис. 7.26, заключается в том, что вся анимация персонажа хранится в одном шаблоне и, следовательно, в одном файле. Единственная неприятность в том, что этот шаблон еще нужно составить! Для этого может понадобиться много времени, не говоря уже о проблеме выравнивания, потому что шаблон создается из ячеек одинакового размера, скажем, $m \times n$ (обычно m и n — это степени двойки), а вокруг каждой ячейки создается рамка толщиной в один пиксель. Затем нужно написать программу, предназначенную для извлечения изображений из отдельных ячеек (размер ячейки и ее другие параметры известны). В зависимости от типа игры и количества используемой в ней графики, применяется тот или иной метод извлечения изображений из шаблона. В любом случае через некоторое время мы напишем функцию, с помощью которой можно извлекать изображения из загружаемых битовых образов, хранящихся в виде отдельных изображений или в виде шаблонов, а затем загружать эти данные в поверхности DirectDraw. Такая программа позволит пользоваться блиттером, однако мы вернемся к этому вопросу позже. А сейчас, чтобы поближе познакомиться с функцией Load_Bitmap_File(), загрузим с ее помощью 8-, 16- и 24-битовый образы и отобразим их в первичном буфере.

Рис. 7.26. Шаблоны битовых образов облегчают доступ и извлечение

НА ЗАМЕТКУ

Большинство изображений, фигурирующих в демонстрационных примерах, были созданы лично мною, а некоторые — другими художниками. Для этого применялись различные графические редакторы и программы моделирования трехмерной графики. Результат сохранялся в формате .bmp. Для таких целей оказываются полезными хорошие графические редакторы, поддерживающие большое количество форматов. Автор обычно пользуется программой Paint Shop Pro — одной из тех, в которых оптимальным образом сочетаются цена и качество.

Загрузка 8-битовых образов

Чтобы загрузить 8-битовое изображение, можно воспользоваться приведенным ниже кодом.

```
BITMAP_FILE bitmap; // Здесь будет храниться битовый образ

if (!(Load_Bitmap_File(&bitmap, "d:\\filename.bmp")))
   { /* ошибка */ }

// Необходимые действия с данными,
// которые хранятся в bitmap.buffer

// Палитра хранится в переменной-члене bitmap.palette

// По окончании необходимо очистить буфер,
// содержащий битовый образ
Unload_Bitmap(&bitmap);
```

Единственная интересная особенность загрузки 8-битовых образов заключается в том, что в структуре BITMAP_FILE содержится информация о палитре. Эту информацию можно использовать для изменения и сохранения палитры DirectDraw, а также для других операций. Это позволяет лучше понять технологию разработки 8-битовых игр.

ВНИМАНИЕ

В рассматриваемом режиме на экран можно вывести только 256 цветов. Поэтому при разработке графики следует подобрать 256-цветную палитру, в которой все изображения, преобразованные к 256 цветам (замечательным инструментом для этого является программа Debabelizer), будут выглядеть достаточно хорошо. Во многих случаях может понадобиться несколько палитр — каждая для своего уровня игры. Однако все изображения в пределах одного и того же уровня должны быть выдержаны в одной палитре!

Теперь настало время продемонстрировать то, с чем не было повода познакомиться до сих пор, а именно изменение записей палитры после того, как палитра уже создана и присоединена к 8-битовой поверхности DirectDraw. В большинстве рассмотренных демонстрационных примеров, работающих в 8-битовых режимах, палитры обычно создавались из случайных или градиентных цветов. Однако на этот раз загружается изображение со своей собственной палитрой. Допустим, эту палитру DirectDraw нужно обновить другими записями. Чтобы выполнить изменение палитры, нужно воспользоваться функцией IDIRECTDRAWPALETTE:SetEntries() (см. приведенный ниже код).

```
BITMAP_FILE bitmap; // 8-битовое изображение

// Предположим, что загружено 8-битовое изображение

if (FAILED(lpddpal->SetEntries(0,0,MAX_COLORS_PALETTE,
  bitmap.palette)))
{ /* ошибка */ }
```

Это так просто, что даже обидно!

В качестве иллюстрации загрузки 8-битовго изображения и вывода его на экран рассмотрим демонстрационную программу DEMO7_10.CPP на прилагаемом компакт-диске. Эта программа загружает 8-битовое изображение в режиме 640×480 и переносит его в первичный буфер.

Загрузка 16-битовых образов

Загружаются 16-битовые образы практически так же, как и 8-битовые, однако при этом не нужно беспокоиться о палитре ввиду ее отсутствия. Кроме того, создавать 16-битовые файлы в формате .bmp позволяют очень мало графических редакторов. Поэтому, чтобы использовать 16-битовый режим DirectDraw, может понадобиться сначала загрузить 24-битовый образ с последующим преобразованием его в 16-битовый с помощью алгоритма изменения шкалы цветности. Чтобы преобразовать 24-битовое изображение в 16-битовое, выполните такие действия.

1. Создайте буфер для 16-битовых значений типа WORD размерностью m×n. В этом буфере будет храниться получившееся в результате изображение.
2. После загрузки 24-битового изображения в структуру BITMAP_FILE осуществите доступ к буферу изображения и преобразуйте каждый 24-битовый пиксель в 16-битовый с помощью такого кода:

```
// Каждый пиксель в буфере BITMAP_FILE.buffer[] закодирован
// как 3-байтовая последовательность интенсивностей синего,
// зеленого и красного цветов (именно в таком порядке)

// Предполагается, что переменная index указывает
// на очередной пиксель...
```

```
UCHAR blue  = (bitmap.buffer[index*3 + 0]) >> 3,
    green = (bitmap.buffer[index*3 + 1]) >> 2,
    red   = (bitmap.buffer[index*3 + 2]) >> 3;

// Создание 16-битового цветового значения
USHORT color = _RGB16BIT565(red,green,blue);
```

Затем значение цвета, хранящееся в переменной color, записывается в предназначенный для этого 16-битовый буфер. Ознакомившись со всеми библиотечными функциями, вы встретите среди них функцию для преобразования 24-битовых изображений в 16-битовые.

Если предположить, что битовый образ имеет 16-битовый формат и никакое преобразование не требуется, то загрузка этого битового образа идентична загрузке 8-битового образа. В качестве примера можно рассмотреть демонстрационную программу DEMO7_11.CPP на прилагаемом компакт-диске, загружающую 24-битовое изображение, преобразующую его в 16-битовое и осуществляющую перенос полученного изображения в первичный буфер.

Загрузка 24-битовых образов

Проще всего загружать 24-битовые образы. Сначала создадим файл с 24-битовым изображением, а затем загрузим его с помощью функции Load_Bitmap_File(). После этого в буфере BITMAP_FILE.buffer[] будут содержаться данные в виде трехбайтовых пикселей, расположенных слева направо, строка за строкой, в формате BGR (синий, зеленый, красный). Об этом нужно помнить, так как порядок следования данных очень важен при их извлечении и использовании! Кроме того, многие графические карты поддерживают не 24-битовый режим, а 32-битовый, позволяющий избежать адресации с использованием нечетных байтов. С помощью дополнительного четвертого байта выполняется выравнивание данных путем заполнения свободных мест незначащей информацией или определения прозрачности цветов. Как бы там ни было, при считывании каждого пикселя из буфера BITMAP_FILE.buffer[] и его записи в первичную поверхность или в 32-битовую внеэкранную поверхность DirectDraw это заполнение нужно выполнять самостоятельно. Ниже приведен пример такого кода.

```
// Каждый пиксель в буфере BITMAP_FILE.buffer[] закодирован
// в виде 3-байтовой последовательности в порядке BGR
// (синий, зеленый, красный)

// Предположим, что переменная index указывает
// на очередной пиксель...
UCHAR blue  = (bitmap.buffer[index*3 + 0]),
    green = (bitmap.buffer[index*3 + 1]),
    red   = (bitmap.buffer[index*3 + 2]);

// Построение 32-битового цветового значения
// в формате A.8.8.8
_RGB32BIT(0,red,green,blue);
```

Этот макрос нам уже встречался, так что не пугайтесь. Освежим память и приведем его еще раз.

```
// Построение 32-битового цветового значения
// в формате A.8.8.8
#define _RGB32BIT(a,r,g,b) ((b) + ((g) << 8) + \
            ((r) << 16) + ((a) << 24))
```

В качестве примера загрузки и отображения симпатичной 24-битовой картинки рассмотрите программу DEMO7_12.CPP на прилагаемом компакт-диске. В этой программе загружается полный 24-битовый образ, устанавливается 32-битовый режим дисплея, после чего изображение копируется в первичную поверхность. Здорово, не так ли?

Заключительное слово по поводу битовых образов

Итак, вы ознакомились с загрузкой битовых образов в 8-, 16- или 24-битовом формате. В результате стало очевидным, что предстоит написать большое количество вспомогательных функций, чем мы и займемся в ближайшем будущем. Кроме того, полезно научиться загружать файлы в формате Targa (с расширением .TGA), так как некоторые программы, моделирующие трехмерную графику, способны выводить анимационные последовательности только в файлы с именем filename*nnnn*.tga, где число *nnnn* изменяется в пределах от 0000 до 9999. Возможно, понадобится загружать анимационные последовательности именно в таком виде, поэтому при изучении библиотечных функций будет продемонстрирована загрузка файлов .tga. Файлы в этом формате загружать намного легче, чем в формате .bmp.

Внеэкранные поверхности

Весь смысл технологии DirectDraw заключается в том, чтобы воспользоваться преимуществами аппаратного ускорения. Увы, это невозможно без применения структур данных и объектов DirectDraw, предназначенных для хранения битовых образов. Поверхности DirectDraw — вот ключ для использования блиттера. Мы уже научились создавать поверхности и задние буферы, а также формировать из них цепочку сменяющих друг друга анимационных страниц. Однако нам еще предстоит узнать, как создавать общие внеэкранные поверхности размерами *m*×*n* в системной или видеопамяти. Располагая такими поверхностями, можно заполнять их битовыми образами, а затем пересылать эти поверхности на экран с помощью блиттера.

В настоящий момент вы уже можете загружать изображения и извлекать из них отдельные биты, так что главная задача решена. Не хватает лишь умения создавать общие внеэкранные поверхности DirectDraw, которые не являются ни первичными поверхностями, ни задним буфером.

Создание внеэкранных поверхностей

Создание внеэкранной поверхности почти идентично созданию первичного буфера, за исключением трех отличий.

1. В поле DDSURFACEDESC2.dwFlags необходимо установить значение DDSD_CAPS|DDSD_WIDTH|DDSD_HEIGHT.
2. В полях DDSURFACEDESC2.dwWidth и DDSURFACEDESC2.dwHeight необходимо задать размеры запрашиваемой поверхности.
3. В поле DDSURFACEDESC2.ddsCaps.dwCaps следует задать значение DDSCAPS_OFFSCREENPLAIN|memory_flags, где с помощью переменной memory_flags определяется место создания поверхности. Если присвоить этой переменной значение DDSCAPS_VIDEOMEMORY, поверхность будет создана в видеопамяти (при наличии там свободного места); если же присвоить ей значение DDSCAPS_SYSTEMMEMORY — то в системной памяти. В последнем случае блиттер практически не будет использоваться, потому что данные битового образа должны передаваться через системную шину.

В качестве примера приведем функцию, с помощью которой можно создавать поверхности любого указанного типа.

```
LPDIRECTDRAWSURFACE7 DDraw_Create_Surface(int width,
                    int height, int mem_flags)
{
// Эта функция создает внеэкранную простую поверхность

DDSURFACEDESC2 ddsd;            // Рабочее описание
LPDIRECTDRAWSURFACE7 lpdds; // Временная поверхность

// Инициализация структуры
DDRAW_INIT_STRUCT(ddsd);

// Установка доступа к полям, задающим ширину,
// высоту и другие свойства поверхности
ddsd.dwFlags = DDSD_CAPS| DDSD_WIDTH | DDSD_HEIGHT;

// Задание размеров новой битовой поверхности
ddsd.dwWidth  = width;
ddsd.dwHeight = height;

// Задание вида поверхности как простой внеэкранной
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN| mem_flags;

// Создание поверхности
if (FAILED(lpdd->CreateSurface(&ddsd,&lpdds,NULL)))
   return(NULL);

// Присвоение цветовому ключу значения 0
DDCOLORKEY color_key; // Используется для установки
                      // цветового ключа
color_key.dwColorSpaceLowValue  = 0;
color_key.dwColorSpaceHighValue = 0;

// Задаем цветовой ключ для источника блиттинга
lpdds->SetColorKey(DDCKEY_SRCBLT, &color_key);

// Возврат поверхности
return(lpdds);

} // DDraw_Create_Surface
```

Теперь посмотрите на код и попытайтесь найти что-нибудь, касающееся пикселей или глубины цвета. Не находите? Странно, не правда ли? Оказывается, не совсем... Отсутствие этой информации объясняется тем, что создаваемая поверхность должна быть совместима с первичной поверхностью, поэтому глубины цветов этих двух поверхностей совпадают и передача этих данных была бы излишней. Итак, чтобы создать в видеопамяти поверхность размером 64×64 пикселей, можно воспользоваться таким кодом:

```
LPDIRECTDRAWSURFACE7 space_ship = NULL; // Здесь хранитсяповерхность
// Создание поверхности
if (!(space_ship = DDraw_Create_Surface(64,64,
```

```
        DDSCAPS_VIDEOMEMORY)))
  { /* ошибка */ }
```

СЕКРЕТ

Создавая поверхности для хранения битовых образов, помещайте в видеопамять в первую очередь те поверхности, в которые предполагается интенсивный вывод графики. Кроме того, рационально сначала создавать самые большие поверхности, постепенно двигаясь в сторону уменьшения их величины.

Теперь с поверхностями можно делать все, что угодно. Например, кто-то захочет заблокировать поверхность, чтобы скопировать в нее битовый образ. Вот как это делается:

```
DDSURFACEDESC2 ddsd;    // Описание поверхности DirectDraw

// Инициализация структуры
DDRAW_INIT_STRUCT(ddsd);

// Блокирование поверхности, проверка наличия ошибок
space_ship->Lock(NULL, &ddsd,
          DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR,
          NULL);

// Выполнение нужных действий с ddsd.lpSurface и ddsd.lPitch

// Снятие блокировки
space_ship->Unlock(NULL);
```

После выполнения всех операций с поверхностью (выход из игры или другие события) ее необходимо освободить. Как обычно, это делается с помощью функции Release():

```
if (space_ship)
  space_ship->Release();
```

Вот и все, что следует знать о создании внеэкранных поверхностей в технологии DirectDraw. Теперь рассмотрим, как выполнять блиттинг поверхностей, например как пересылать задний буфер или первичную поверхность.

Блиттинг внеэкранных поверхностей

Теперь, когда вы научились загружать битовые образы, создавать поверхности и пользоваться блиттером, пришло время собрать это все вместе и выполнить настоящую анимацию! В этом разделе рассматривается, как загрузить битовые образы, содержащие кадры анимации какого-то объекта или персонажа (это может быть все, что угодно, например корабль или животное), создать несколько небольших поверхностей, предназначенных для хранения каждого кадра анимации, а затем загрузить изображения в каждую их этих поверхностей. После загрузки битовых образов в поверхности возникает естественное желание реализовать их пересылку на экран и анимировать объект!

Фактически вы уже знаете, как все это выполнить. Неизвестно лишь, как использовать блиттер для пересылки изображения из какой-то поверхности, отличной от заднего буфера, в первичный буфер. Однако нет никакой разницы, осуществляется ли блиттинг из заднего буфера или из вторичной поверхности другого вида. На рис. 7.27 изображено несколько небольших поверхностей, каждая из которых — это отдельный анимационный кадр. Кроме того, на этом рисунке показаны первичная поверхность и поверхность заднего буфера. План состоит в том, чтобы загрузить все эти битовые образы в маленькие поверхности (создав объект для анимации), организовать пересылку этих поверхностей с помощью блиттера в задний буфер и

получить результат путем переключения страниц. Далее периодически осуществляется блиттинг очередного изображения, причем целевое положение каждый раз слегка сдвигается, чтобы анимированный объект перемещался по экрану.

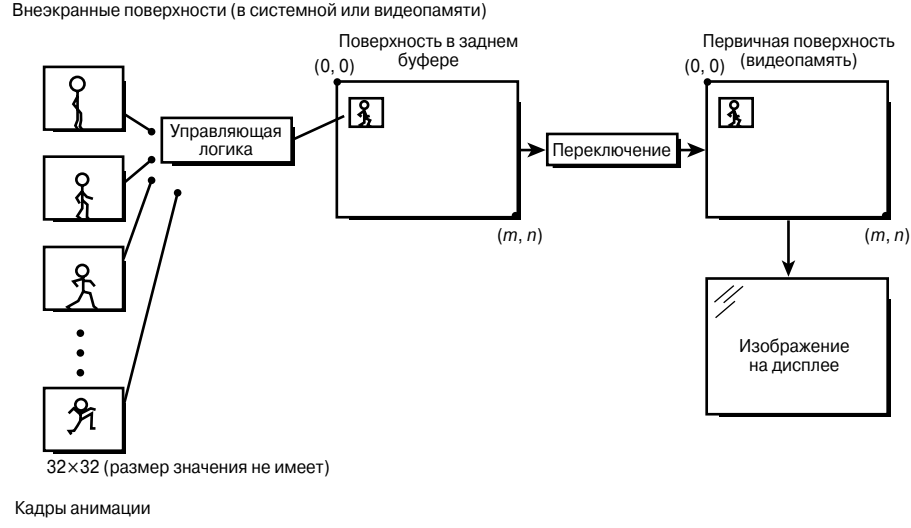


Рис. 7.27. Блиттинг внеэкранных поверхностей в задний буфер

Настройка блиттера

Чтобы настроить блиттер, выполните такие действия.

1. Задайте исходную прямоугольную область, из которой будет выполняться блиттинг. Скорее всего, это будет маленькая поверхность (8×8, 16×16, 64×64 и т.д.), содержащая необходимое изображение. Как правило, координаты задаются от (0,0) до (*ширина*-1,*высота*-1), т.е. вся поверхность.
2. Задайте прямоугольную область-получатель, в роли которой обычно выступает задний буфер. На этом этапе могут встретиться определенные трудности, связанные с правильным расположением исходного изображения: следует помнить, что если левый верхний угол изображения имеет координаты (x,y), то его правый нижний угол расположен в точке (x+*ширина*-1, y+*высота*-1).
3. Вызовите функцию `IDIRECTDRAWSURFACE7::Blt()` с нужными параметрами. Как это сделать, будет показано ниже.

НА ЗАМЕТКУ

Если прямоугольную поверхность назначения задать больше или меньше исходной, блиттер изменит масштаб изображения соответствующим образом. Этот принцип положен в основу 2,5D-игр с масштабируемыми спрайтами.

Прежде чем приступить к использованию функции `Blt()`, необходимо рассмотреть еще одну тему — работу с цветовыми ключами.

Цветовые ключи

Объяснение принципов работы с цветовыми ключами несколько затруднено тем, что в DirectDraw отсутствует устоявшаяся терминология по этому вопросу. Тем не менее попытаемся изложить этот материал. При выполнении операций с битовыми образами в

большинстве случаев пересылаются образы прямоугольной формы. Однако, если битовый образ содержит небольшое изображение монстра, обычно нужно скопировать не весь прямоугольник, а только те биты, из которых состоит сам монстр. Чтобы добиться этого эффекта, цвет одного фонового цвета (или нескольких фоновых цветов) нужно задать как прозрачный. На рис. 7.28 проиллюстрирован блиттинг с прозрачными цветами и без них. Этот вопрос уже рассматривался, и даже в качестве упражнения был реализован программный блиттер.

Рис. 7.28. Блиттинг с прозрачными цветами (вверху) и без них (внизу)

Технология DirectDraw содержит значительно более сложную систему цветовых ключей, которая не сводится к обычному выбору прозрачного цвета. В этой схеме можно выполнять намного больше действий, чем обычный блиттинг с основными прозрачными

цветами. Приведем беглый обзор различных типов цветовых ключей, а затем научимся задавать их для интересующих видов операций.

Выбор цветовых ключей источника

В этом разделе пойдет речь о выборе цветовых ключей, которые вы хотите использовать в исходном изображении. Здесь нет ничего сложного. По сути, выбирается один цветовой индекс (в 256-цветном режиме) или диапазон цветовых значений в формате RGB, которые в исходном изображении будут выступать как прозрачные. Затем во время блиттинга исходного изображения те пиксели, цветовые значения которых заданы как прозрачные, в место назначения не копируются. Этот процесс схематически проиллюстрирован на рис. 7.14. Цветовой ключ поверхности можно задавать как во время создания этой поверхности, так и позже с помощью функции IDIRECTDRAWSURFACE7::SetColorKey(). Ниже будут показаны оба метода, однако сначала рассмотрим структуру данных DDCOLORKEY, в которой хранятся цветовые ключи.

```
typedef struct _DDCOLORKEY
{
  DWORD dwColorSpaceLowValue;  // Нижнее значение (включительно)
  DWORD dwColorSpaceHighValue; // Верхнее значение (включительно)
} DDCOLORKEY,FAR* LPDDCOLORKEY;
```

Если речь идет о 8-битовых поверхностях, эти значения должны быть цветовыми индексами. Если же используются 16-, 24- или 32-битовые поверхности, в роли нижнего и верхнего значений цветовых ключей выступают значения типа WORD в кодировке RGB. Например, предположим что мы работаем в 8-битовом режиме, и нужно, чтобы цветовой индекс 0 был прозрачным. Вот как можно было бы задать цветовой ключ в этом случае:

```
DDCOLORKEY key;

key.dwColorSpaceLowValue  = 0;
key.dwColorSpaceHighValue = 0;
```

Если же необходимо, чтобы прозрачные цвета задавались цветовыми индексами в диапазоне от 10 до 20 (включительно), используется такой код:

```
key.dwColorSpaceLowValue  = 10;
key.dwColorSpaceHighValue = 20;
```

Теперь предположим, что установлен 16-битовый режим с кодировкой 5.6.5 и в роли прозрачного будет выступать синий цвет:

```
key.dwColorSpaceLowValue  = _RGB16BIT565(0,0,32);
key.dwColorSpaceHighValue = _RGB16BIT565(0,0,32);
```

Аналогично, если в том же 16-битовом режиме прозрачными должны быть цвета в диапазоне от черного до красного (половинной интенсивности), можно воспользоваться таким кодом:

```
key.dwColorSpaceLowValue  = _RGB16BIT565(0,0,0);
key.dwColorSpaceHighValue = _RGB16BIT565(16,0,0);
```

Идея понятна? Теперь рассмотрим, как задаются цветовые ключи поверхности DirectDraw во время ее создания. Все, что нужно для этого сделать, — добавить в поле описания поверхности dwFlags флаг DDSD_CKSRCBLT (список возможных значений этого поля приведен в табл. 7.5), а затем присвоить значения нижнего и верхнего цветовых ключей полям DDSURFACEDESC2.ddckCKSrcBlt.dwColorSpaceLowValue и DDSUR-

FACEDESC2.ddckCKSrcBlt.dwColorSpaceHighValue (имеются также члены структуры, соответствующие цветовым ключам поверхности назначения и наложения).

Таблица 7.5. Флаги цветовых ключей поверхности

Значение	*Описание*
DDSD_CKSRCBLT	Указывает, что член ddckCKSrcBlt структуры DDSURFACEDESC2 является корректным и содержит информацию о цветовых ключах исходной поверхности
DDSD_CKDESTBLT	Указывает, что член ddckCKDestBlt структуры DDSURFACEDESC2 является корректным и содержит информацию о цветовых ключах поверхности назначения
DDSD_CKDESTOVERLAY	Указывает, что член ddckCKDestOverlay структуры DDSURFACEDESC2 является корректным и содержит информацию о целевых цветовых ключах наложения
DDSD_CKSRCOVERLAY	Указывает, что член ddckCKDestOverlay структуры DDSURFACEDESC2 является корректным и содержит информацию об исходных цветовых ключах наложения

Приведем пример:

```
DDSURFACEDESC2 ddsd;      // Рабочее описание
LPDIRECTDRAWSURFACE7 lpdds; // Временная поверхность

// Инициализация структуры
DDRAW_INIT_STRUCT(ddsd);

// Установка доступа к полям, задающим ширину,
// высоту и другие свойства поверхности
ddsd.dwFlags = DDSD_CAPS| DDSD_WIDTH | DDSD_HEIGHT | DDSD_CKSRCBLT;

// Задание размеров новой растровой поверхности
ddsd.dwWidth  = width;
ddsd.dwHeight = height;

// Задание вида поверхности как простой внеэкранной
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN| mem_flags;

// Задание полей цветовых ключей
ddsd.ddckCKScrBlt.dwColorSpaceLowValue  = low_color;
ddsd.ddckCKScrBlt.dwColorSpaceHighValue = high_color;

// Создание поверхности
if (FAILED(lpdd->CreateSurface(&ddsd,&lpdds,NULL)))
  return(NULL);
```

Как бы ни была создана поверхность — с применением цветового ключа или без него, цветовой ключ можно задать в любой момент с помощью функции IDIRECTDRAWSURFACE7:SetColorKey():

```
HRESULT SetColorKey(DWORD dwFlags,
          LPDDCOLORKEY lpDDColorKey);
```

Список возможных флагов этой функции приведен в табл. 7.6.

Таблица 7.6. Флаги функции SetColorKey()

Значение	*Описание*
DDCKEY_COLORSPACE	Указывает, что структура содержит цветовое пространство. Этот флаг должен быть установлен, если задается диапазон цветов
DDCKEY_SRCBLT	Указывает, что при блиттинге заданные в структуре цветовой ключ или цветовое пространство будут использоваться в качестве цветового ключа для исходной поверхности
DDCKEY_DESTBLT	Указывает, что при блиттинге заданные в структуре цветовой ключ или цветовое пространство будут использоваться в качестве цветового ключа для поверхности назначения
DDCKEY_DESTOVERLAY	Этот флаг устанавливается, если в структуре задается цветовой ключ или цветовое пространство для использования в качестве цветового ключа поверхности назначения при операциях с наложением (для опытных программистов)
DDCKEY_SRCOVERLAY	Этот флаг устанавливается, если в структуре задается цветовой ключ или цветовое пространство для использования в качестве цветового ключа исходной поверхности при операциях с наложением (для опытных программистов)

Приведем пример:

```
// Считаем, что lpdds указывает
// на корректную поверхность

// Задание цветового ключа
DDCOLORKEY color_key; // Применяется для задания
                      // цветового ключа
color_key.dwColorSpaceLowValue  = low_value;
color_key.dwColorSpaceHighValue = high_value;

// Установка цветового флага для исходной поверхности
// блиттинга; обратите внимание на использование
// флага DDCKEY_SRCBLT
lpdds->SetColorKey(DDCKEY_SRCBLT, &color_key);
```

НА ЗАМЕТКУ

Если в качестве цветового ключа задан диапазон цветов, при вызове функции SetColorKey() необходимо воспользоваться флагом DDCKEY_COLORSPACE, например:

```
lpdds->SetColorKey(DDCKEY_SRCBLT | DDCKEY_COLORSPACE,
                   &color_key);
```

В противном случае DirectDraw сузит диапазон до одного значения.

Выбор цветовых ключей поверхности назначения

В основе применения цветовых ключей поверхности назначения лежит целая сложная теория, однако, по-видимому, она никогда не будет использована в полной мере. Основная концепция выбора цветовых ключей назначения проиллюстрирована на рис. 7.29. Идея такова: в поверхности назначения задается цвет или диапазон цветов, в который может выполняться блиттинг. По сути, создается маска для объектов разных видов. Таким путем можно имитировать окна, заборы и т.д.

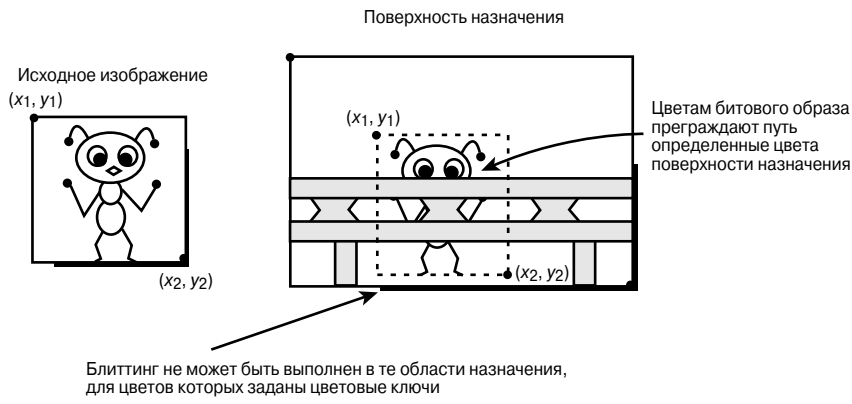


Рис. 7.29. Цветовые ключи поверхности назначения

Цветовые ключи поверхности назначения можно задавать так же, как и для исходной поверхности, за исключением нескольких флагов. Например, чтобы задать цветовой ключ поверхности назначения в процессе создания поверхности, в качестве значения флага DDRAWSURFACEDESC2.dwFlags нужно использовать не DDSD_CKSRCBLT, а DDSD_CKDESTBLT. Кроме того, для этого случая подойдут значения ключа ddsd.ddckCKDestBlt, а не ddsd.ddckCKSrcBlt:

```
// настройка полей цветового ключа
ddsd.ddckCKDestBlt.dwColorSpaceLowValue  = low_color;
ddsd.ddckCKDestBlt.dwColorSpaceHighValue = high_color;
```

Если цветовой ключ поверхности назначения задается после ее создания, нужно воспользоваться функцией SetColorKey() точно так же, как и для исходной поверхности, однако при этом вместо флага DDCKEY_SRCBLT следует установить флаг DDCKEY_DESTBLT:

```
lpdds->SetColorKey(DDCKEY_DESTBLT, &color_key);
```

ВНИМАНИЕ

В настоящее время цветовые ключи поверхности назначения поддерживаются только на уровне аппаратных абстракций (HAL), но не на уровне эмуляции аппаратных средств (HEL). Таким образом, цветовые ключи поверхности назначения не работают без аппаратной поддержки. Возможно, эта ситуация изменится в последующих версиях DirectX.

Наконец, есть еще два типа ключей: исходных наложений и наложений назначения. Для наших целей они не нужны, однако оказываются полезными в работе с видеопроцессором. Более подробную информацию можно найти в документации DirectX SDK.

Использование блиттера (наконец-то!)

Теперь, когда со всеми предварительными приготовлениями покончено, блиттинг внеэкранной поверхности в любую другую — дело одной минуты. Вот как это делается. Предположим, что создана поверхность с изображением размером 64×64 и 8-битовыми цветами, в которой цветовой индекс 0 задан как прозрачный:

```
DDSURFACEDESC2 ddsd;                  // Рабочее описание
LPDIRECTDRAWSURFACE7 lpdds_image; // Временная поверхность

// Инициализация структуры
DDRAW_INIT_STRUCT(ddsd);
```

```
// Установка доступа к полям, задающим ширину,
// высоту и другие свойства поверхности
ddsd.dwFlags = DDSD_CAPS| DDSD_WIDTH | DDSD_HEIGHT |
        DDSD_CKSRCBLT;

// Задание размеров новой растровой поверхности
ddsd.dwWidth  = 64;
ddsd.dwHeight = 64;

// Задание вида поверхности как простой внеэкранной
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN| mem_flags;

// Задание полей цветовых ключей
ddsd.ddckCKScrBlt.dwColorSpaceLowValue  = 0;
ddsd.ddckCKScrBlt.dwColorSpaceHighValue = 0;

// Создание поверхности
if (FAILED(lpdd->CreateSurface(&ddsd,&lpdds_image,NULL)))
  return(NULL);
```

Далее предположим, что создана первичная поверхность (lpddsprimary) и поверхность заднего буфера (lpddsback), причем поверхность lpdds_image нужно переслать в точку (x, y) заднего буфера с заданным цветовым ключом исходной поверхности. Для этого можно воспользоваться приведенным ниже кодом.

```
// Прямоугольная область назначения
dest_rect.left  = x;
dest_rect.top   = y;
dest_rect.right  = x+64-1;
dest_rect.bottom = y+64-1;

// Исходная прямоугольная область
source_rect.left  = 0;
source_rect.top   = 0;
source_rect.right  = 64-1;
source_rect.bottom = 64-1;

// Блиттинг в поверхность назначения
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
      &source_rect,
      (DDBLT_WAIT | DDBLT_KEYSRC),
      NULL)))
  return(0);
```

Вот и все! Обратите внимание на флаг DDBLT_KEYSRC. При вызове блиттера его обязательно нужно установить. В противном случае цветовой ключ не работает, даже если он задан для данной поверхности.

ВНИМАНИЕ

Выполняя блиттинг, следует позаботиться об отсечении. Если вы забудете задать в поверхности назначения область отсечения или выполните блиттинг за рамки этой области, последствия могут оказаться плачевными. Все, что нужно для этого сделать, — вызвать функцию DDraw_Attach_Clipper() и задать единую прямоугольную область отсечения, границы которой совпадают с границами экрана.

Теперь все готово для того, чтобы ознакомиться с довольно красивой демонстрационной программой DEMO7_13.CPP на прилагаемом компакт-диске. На рис. 7.30 приведена копия экрана этой программы. Я решил добавить в этот пример некоторые игровые элементы, поэтому из него можно получить больше, чем обычное перемещение битовых образов. Суть происходящего в том, что в этой программе загружается большой фоновый битовый образ, а также несколько кадров анимации, чтобы "оживить" инопланетянина. При формировании каждого кадра в задний буфер копируется фоновое изображение, а также несколько анимационных копий инопланетянина (каждая из которых представляет собой отдельную поверхность). Затем осуществляется анимация и перемещение инопланетян. Подумайте над тем, каким образом в программу можно добавить персонаж, управляемый игроком с помощью клавиатуры.

Вращение и масштабирование битовых образов

В технологии DirectDraw поддерживается вращение и масштабирование битовых образов (рис. 7.31). Однако вращение поддерживается только на уровне аппаратных абстракций. Это значит, что если отсутствует аппаратная поддержка вращения, то вам не повезло. Может возникнуть вопрос: "Почему это на уровне эмуляции аппаратных средств масштабирование поддерживается, а вращение — нет?" Дело в том, что вращение битовых образов происходит примерно в 10—100 раз медленнее, чем масштабирование. Поэтому программисты Microsoft решили, что программная реализация вращения будет работать слишком медленно, независимо от того, насколько хороший код они напишут для этой операции! Короче говоря, всегда можно рассчитывать на масштабирование, но не на вращение. Всегда можно самостоятельно написать функцию, с помощью которой осуществляется вращение битовых образов, однако это достаточно сложно, и в играх такая необходимость возникает редко. Во всяком случае в данной книге эта тема не рассматривается. Однако следует заметить, что отсутствие программной поддержки вращения послужило одной из причин того, что компания Microsoft объединила компоненты DirectDraw и Direct3D. Если все элементы игры моделируются с помощью трехмерных многоугольников, а текстура отображает на них ваши двухмерные битовые образы и помещает камеру обзора в произвольное место пространства, то тем самым эффективно осуществляется вращение и сдвиг, а также достигаются прозрачность, световые и другие эффекты. Однако при этом вывод двухмерной графики происходит за счет аппаратных устройств трехмерной графики. Все бы ничего, но кому-то такой подход может и не понравиться.

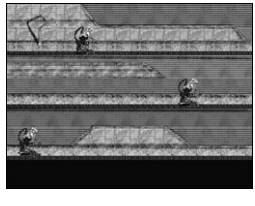


Рис. 7.30. Программа DEMO7_13.EXE в действии

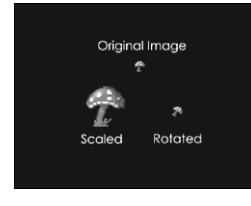


Рис. 7.31. Масштабирование и вращение битовых образов

Масштабирование битовых образов выполняется легко. Все, что для этого нужно, — изменить размеры прямоугольной области назначения, чтобы они отличались от размеров исходного изображения. В результате над исходным изображением будет выполнено

преобразование масштабирования. Например, предположим, что у нас есть изображение 64×64 пикселя, которое нужно масштабировать к размеру *m*×*n* и поместить в точку с координатами (x, y). Это выполняется с помощью такого кода:

```
// Заполнение прямоугольной области назначения
dest_rect.left   = x;
dest_rect.top    = y;
dest_rect.right  = x+m-1;
dest_rect.bottom = y+n-1;

// Заполнение исходной прямоугольной области
source_rect.left   = 0;
source_rect.top    = 0;
source_rect.right  = 64-1;
source_rect.bottom = 64-1;

// Блиттинг в поверхность назначения
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
       &source_rect,
       (DDBLT_WAIT | DDBLT_KEYSRC),
       NULL)))
  return(0);
```

Все довольно просто. Выполнить вращение несколько сложнее, так как нужно задать данные структуры DDBLTFX. Чтобы выполнить операцию вращения битового образа, необходимо иметь аппаратное ускорение, поддерживающее эту операцию (что встречается крайне редко), а также задать структуру DDBLTFX. Ниже показано, как это делается.

```
DDBLTFX ddbltfx; // Здесь хранятся данные

// Инициализация структуры
DDRAW_INIT_STRUCT(ddbltfx);

// Задание угла вращения в единицах, равных 1/100 градуса
ddbltfx.dwRotationAngle = angle;
```

После этого обычным образом вызывается функция Blt(), однако при этом к параметрам-флагам добавляется флаг DDBLT_ROTATIONANGLE и параметр ddbltfx.

```
// блиттинг в поверхность назначения
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
       &source_rect,
      (DDBLT_WAIT | DDBLT_KEYSRC | DDBLT_ROTATIONANGLE),
       &ddbltfx)))
  return(0);
```

НА ЗАМЕТКУ

Чтобы определить, поддерживается ли аппаратное вращение, следует запросить свойства поверхности и проверить наличие в члене dwFxCaps структуры DDSCAPS флагов DDFXCAPS_BLTROTATION*. Для запроса свойств поверхности можно воспользоваться функцией IDIRECTDRAWSURFACE7::GetCaps(), что будет описано в конце главы.

Прежде чем перейти к примерам, демонстрирующим масштабирование и вращение в DirectDraw, посвятим немного времени теории дискретизации и ее применению к реализации масштабирования (по крайней мере, на программном уровне).

Теория дискретизации

Этот раздел будет коротким. Работая с битовыми образами, мы на самом деле работаем с сигналами; просто эти сигналы имеют вид дискретных данных, которыми закодировано двухмерное изображение, а не непрерывных аналоговых данных, таких, как радиосигнал. Таким образом, к изображениям можно применять концепции обработки сигналов, точнее, концепции обработки цифровых сигналов. Одна из интересующих нас тем — дискретизация и отображение данных.

В процессе освоения царства двух- и трехмерной графики неоднократно возникают ситуации, в которых нужно выполнить сначала дискретизацию данных растрового изображения, а затем какие-то операции с этими данными, например масштабирование, вращение или текстурное отображение. Следует различать два вида отображений общего вида: *прямые отображения* (forward mapping) и *обратные отображения* (inverse mapping). На рис. 7.32 и 7.33 приведены поясняющие графические схемы.

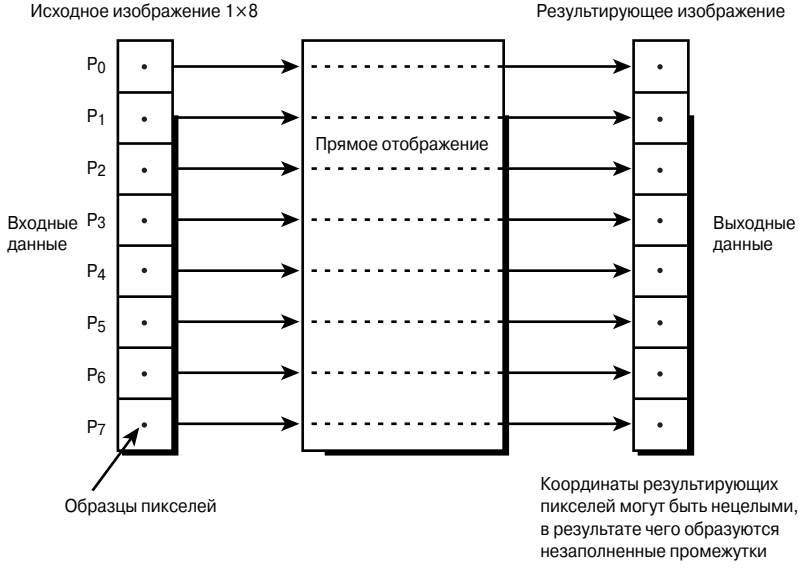


Рис. 7.32. Теория дискретизации: прямое отображение

При прямом отображении пиксели извлекаются из исходного изображения и отображаются (помещаются) в результирующее изображение. Единственная проблема в том, что некоторые исходные пиксели в силу выбора вида отображающей функции могут не иметь своего образа в результирующем изображении.

Обратное отображение работает намного лучше, чем прямое. Его суть заключается в том, что каждому пикселю результирующего изображения сопоставляется определенный пиксель исходного. Конечно же, при этом тоже возникают проблемы: иногда в исходном изображении может быть недостаточно данных, чтобы заполнить все пиксели результирующего изображения, и тогда один и тот же исходный пиксель отображается в несколько результирующих пикселей. В связи с этим возникает проблема ступенчатости краев. Эта же проблема может возникнуть, если данных в исходном изображении слишком много, но ее можно разрешить с помощью усреднения или различных математических фильтров. Лучше все-таки, когда данных слишком много, чем когда их мало.

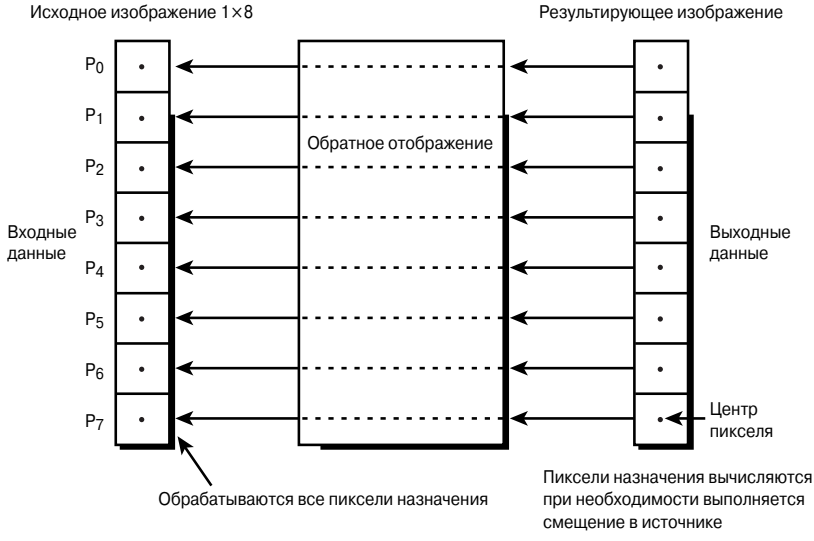


Рис. 7.33. Теория дискретизации: обратное отображение

Масштабирование — это операция, которую можно выполнить как с помощью прямого, так и с помощью обратного отображения. Рассмотрим обратное отображение для масштабирования одномерного битового образа, после чего этот алгоритм можно будет обобщить на два измерения. Это важный момент: во многих алгоритмах обработки изображений направления по различным осям можно считать независимыми. Это означает, что, если известен алгоритм для одномерного изображения, он легко обобщается на произвольное число измерений. При этом результат преобразования вдоль одной из осей не влияет на другие направления.

- **Пример 1.** Пусть битовый образ размером 1×4 нужно преобразовать в битовый образ размером 1×8. Результат показан на рис. 7.34. При этом каждый исходный пиксель копируется в результирующее изображение дважды.

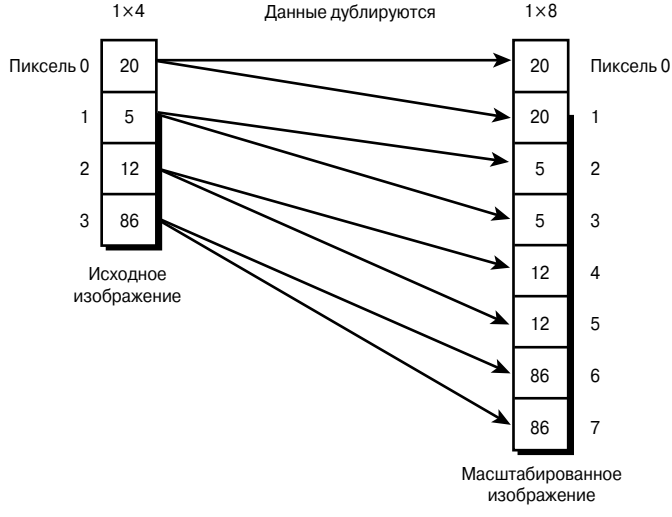


Рис. 7.34. Преобразование битового образа 1×4 пикселя в битовый образ 1×8 пикселей

- **Пример 2.** Пусть битовый образ размером 1×4 нужно преобразовать в битовый образ размером 1×2. Результат показан на рис. 7.35. Два исходных пикселя опущены, в результате чего возникает вопрос: насколько корректна пропажа части информации? Здесь нет однозначного ответа. Данные, несомненно, теряются, но, несмотря на это, метод работает, причем работает быстро.

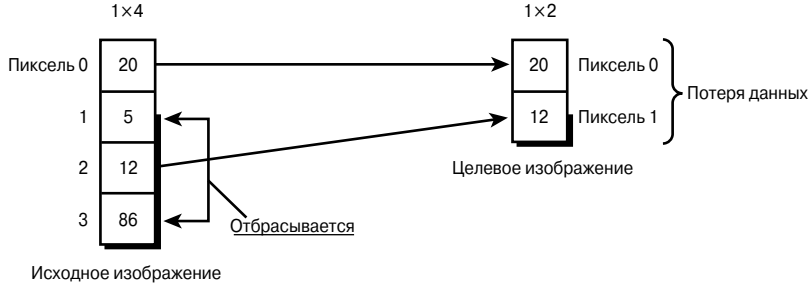


Рис. 7.35. Преобразование битового образа размером 1×4 пикселя в битовый образ размером 1×2 пикселя

В обоих рассмотренных примерах для обработки изображений лучше было бы применить фильтры. Так, в первом примере вместо копирования отдельных пикселей в качестве значения каждого дополнительного пикселя можно было бы использовать усредненные значения пикселей, которые находятся сверху и снизу. При этом растянутое изображение выглядит лучше. Отсюда произошел графический термин *билинейная фильтрация* (bi-linear filtering); она базируется на изложенной выше идее, но применительно к двум измерениям. Во втором примере фильтр можно применить точно таким же образом: усреднить значения пикселей. Тогда, несмотря на выпадение двух пикселей, определенная часть содержащейся в них информации сохраняется в результирующих пикселях и полученное изображение будет выглядеть более естественно.

С фильтрацией данных мы ознакомимся позже, а пока что просто выполним грубое масштабирование “в лоб”. При рассмотрении примеров следует иметь в виду, что исходное изображение дискретизировано с некоторым шагом, на основе которого и заполняется результирующее изображение. На языке формул это выглядит так:

```
// Высота исходного одномерного битового образа
float source_height;

// Высота масштабированного одномерного битового образа
float destination_height;

// Коэффициент преобразования
float sample_rate = source_height/destination_height;

// Индекс исходных данных
float sample_index = 0;

// Построение масштабированного битового образа
for (index = 0; index < dest_height; index++)
{
  // Запись пикселя
  dest_bitmap[index] =
  source_bitmap[(int)sample_index];
```

```
    // Смещение индекса исходных данных
    sample_index+=sample_rate;

} // for index
```

Приведенного выше кода достаточно для выполнения масштабирования. Конечно же, при работе с реальным изображением нужно добавить еще одно измерение. Без вычислений с плавающей точкой можно было бы и обойтись, однако код работает и в таком виде.

Рассмотрим конкретный пример. Пусть исходный битовый образ размером 1×4 имеет такой вид.

Значения пикселей:

```
source_bitmap[0] = 12
source_bitmap[1] = 34
source_bitmap[2] = 56
source_bitmap[3] = 90
```

Выполним масштабирование изображения размером 1×4 пикселя в изображение размером 1×8 пикселей:

```
source_height = 4
dest_height   = 8
sample_rate   = 4/8 = 0.5
```

Результаты работы алгоритма (с учетом округления):

index	sample_index	dest_bitmap[index]
0	0	12
1	0.5	12
2	1.0	34
3	1.5	34
4	2.0	56
5	2.5	56
6	3.0	90
7	3.5	90

Получается не так уж плохо — каждый пиксель просто дублируется. Теперь попытаемся выполнить сжимающее масштабирование, уменьшив высоту исходного изображения до трех пикселей:

```
source_height = 4
dest_height   = 3
sample_rate   = 4/3 = 1.333
```

Результаты работы алгоритма (с учетом округления):

index	sample_index	dest_bitmap[index]
0	0	12
1	1.333	34
2	2.666	56

Обратите внимание, что последний пиксель исходного изображения в результирующем изображении отсутствует. Это может устраивать вас, а может и нет; возможно, кто-то захочет, чтобы при масштабировании верхний и нижний пиксели всегда отображались в результирующее изображение, если его высота не меньше двух пикселей, а вместо них были опущены какие-то промежуточные пиксели. В этой ситуации вступает в игру ок-

ругление и смещение переменных sample_rate и sample_index — поразмышляйте над этим, если возникнет необходимость.

Теперь, когда вы научились выполнять масштабирование изображения самостоятельно, попробуйте сделать это с помощью DirectDraw. Программа DEMO7_14.CPP на прилагаемом компакт-диске — это переработанная программа DEMO7_13.CPP, к которой добавлен код масштабирования фигур инопланетян, чтобы они были разных размеров. Если компьютер обладает аппаратным масштабированием, демонстрационный пример работает очень плавно, в противном случае ухудшение качества будет заметным. Позже в этой главе рассматривается, как с помощью функции IDIRECTDRAWSURFACE7::GetCaps() проверить наличие аппаратной поддержки масштабирования.

Цветовые эффекты

Следующая тема, которую нам предстоит рассмотреть, — цветовая анимация и некоторые трюки, которые выполняются с цветами. Раньше, когда были доступны только 256-цветные режимы с палитрой, было придумано много приемов и методов, в которых применяется эффект мгновенного изменения цветов. Этот эффект заключается в том, что изменение одного или нескольких регистров палитры тут же проявляется на экране. Благодаря быстрому развитию аппаратного обеспечения и аппаратного ускорения время 256-цветных режимов безвозвратно ушло, но знание этих методов по-прежнему актуально. Оно способствует пониманию других близких концепций, не говоря уже о том, что пройдет еще немало времени, прежде чем все игры полностью будут использовать RGB-цвета. Пока у пользователей все еще много компьютеров с процессорами 486, и даже процессоры Pentium с небольшой тактовой частотой способны поддерживать нормальную скорость игр только в 256-цветных режимах.

Цветовая анимация в 256-цветных режимах

В основе цветовой анимации лежат различные операции динамического изменения или сдвига цветовой палитры. Например, манипулируя записями таблицы цветов “на лету”, можно создать мерцающие огни, добиться эффектов горения и многих других. Преимущество такого подхода состоит в том, что с его помощью можно управлять любым отображаемым на экране объектом, манипуляция значениями цвета пикселей которого осуществляется изменением записей в таблице цветов.

Представьте себе, как трудно было бы добиться этого с помощью битовых образов. Например, пусть на экран нужно вывести изображение небольшого космического корабля с мерцающими огоньками. Для этого понадобились бы битовые образы, воспроизводящие каждый кадр анимации. С помощью цветовой анимации добиться нужного эффекта намного проще. Необходимо всего лишь создать один битовый образ, присвоить тем его пикселям, которыми воспроизводятся огоньки, особый цветовой индекс, а затем анимировать только этот цветовой индекс. Этот трюк проиллюстрирован на рис. 7.36.

Два моих любимых эффекта — мерцание и свечение. Сначала рассмотрим функцию для реализации мерцающих огней. Допустим, вы хотите реализовать такие функциональные возможности:

- до 256 мерцающих огоньков;
- каждый огонек имеет цвета, соответствующие включенному и выключенному состояниям, а также время переключения включенных и выключенных состояний;
- включение и выключение мерцающих огоньков и/или сброс их параметров;
- прекращение мерцания каждого огонька и повторное использование массива данных.

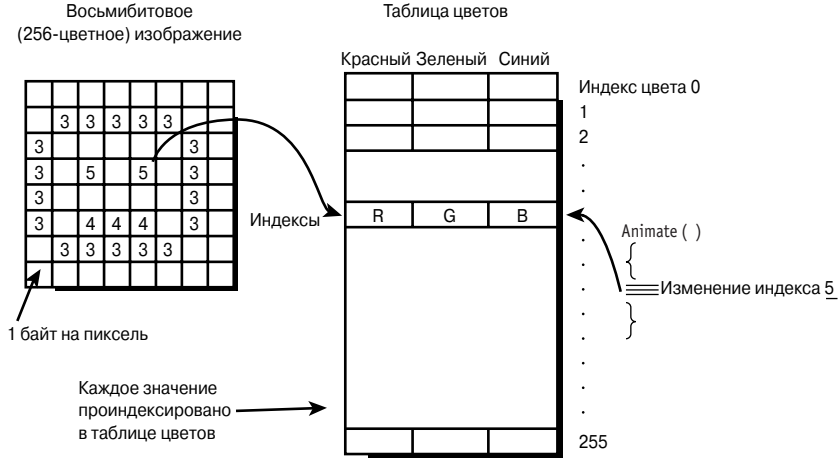


Рис. 7.36. Цветовая анимация путем манипуляции палитрой

Этот пример прекрасно подходит для демонстрации некоторых устоявшихся методов обработки данных и обновления записей палитры DirectDraw. Стратегия, которой мы будем придерживаться, — это создание единой функции, с помощью которой можно создавать и удалять огоньки, а также анимировать их. Эта функция будет использовать локальные статические массивы данных и работать в перечисленных ниже режимах.

BLINKER_ADD. Этот режим применяется для добавления цветов мерцающих огней в базу данных. При вызове функция возвращает номер идентификатора, используемого для обращения к мерцающему огню. В системе может храниться вплоть до 256 таких огней.

BLINKER_DELETE. Удаляет мерцающий огонек с указанным идентификатором.

BLINKER_UPDATE. Обновляет параметры переключения огонька с указанным идентификатором.

BLINKER_RUN. Обрабатывает все огоньки в течение одного цикла.

Ниже приведена структура данных BLINKER, используемая для создания и хранения данных об огне.

```
// Структура данных мерцающего огня
typedef struct BLINKER_TYP
{
  // Значения, задаваемые пользователем

  int color_index;
  // Индекс цвета, соответствующий мерцанию
  PALETTEENTRY on_color;
   // RGB-значение цвета включенного огонька
  PALETTEENTRY off_color;
   // RGB-значение цвета выключенного огонька
  int on_time;
   // Количество кадров, в течение которых огонек включен
  int off_time;
  // Количество кадров, в течение которых огонек выключен


  // Внутренние члены
```

```
  int counter;  // Счетчик состояний
  int state;    // Состояние огонька: -1 — выключен,
                // 1 — включен, 0 — не работает
} BLINKER, *BLINKER_PTR;
```

Сначала заполняются пользовательские поля структуры, после чего вызывается функция с командой BLINKER_ADD. Общий принцип работы такой: для добавления, удаления или обновления огоньков функция вызывается в любой момент, но в режиме BLINKER_RUN это можно делать не более одного раза за кадр. Код функции приводится ниже.

```
int Blink_Colors(int command, BLINKER_PTR new_light, int id)
{
// С помощью этой функции создается набор мерцающих огоньков

static BLINKER lights[256]; // Поддерживает вплоть до 256 огоньков
static int initialized = 0;      // Проверка инициализации функции

// Проверка первого запуска функции
if (!initialized)
{
  // Установка в положение инициализации
  initialized = 1;
  // Обнуление всех структур
  memset((void *)lights,0, sizeof(lights));
} // if

// Проверка команды пользователя
switch (command)
  {
  case BLINKER_ADD: // Добавление огня в базу данных
  {
    // Просмотр базы данных и поиск свободной записи
    for (int index=0; index < 256; index++)
    {
      // Свободна ли запись?
      if (lights[index].state == 0)
      {
        // Задание огонька
        lights[index] = *new_light;
        // Задание внутренних полей
        lights[index].counter =  0;
        lights[index].state   = -1; // Выключен

        // Обновление записи палитры
        lpddpal->SetEntries(0,
              lights[index].color_index,
              1,&lights[index].off_color);

        // Возврат идентификатора
        return(index);
      } // if
    } // for index
  } break;
```

```
case BLINKER_DELETE:
// Удаление огонька с указанным идентификатором
{
   // Удаление огонька с указанным идентификатором
   if (lights[id].state != 0)
   {
   // Удаление огонька
   memset((void *)&lights[id],0,sizeof(BLINKER));
   // Возвращение идентификатора
   return(id);
   } // if
   else
      return(-1); // Проблема
} break;

case BLINKER_UPDATE:
// Обновление огонька с указанным идентификатором
{
   // Проверка состояния огонька
   if (lights[id].state != 0)
   {
      // Обновление только параметров переключения
      lights[id].on_color  = new_light->on_color;
      lights[id].off_color = new_light->off_color;
      lights[id].on_time   = new_light->on_time;
      lights[id].off_time  = new_light->off_time;

      // Обновление записей палитры
       if (lights[id].state == -1)
          lpddpal->SetEntries(0,lights[id].color_index,
                      1,&lights[id].off_color);
       else
          lpddpal->SetEntries(0,lights[id].color_index,
                      1,&lights[id].on_color);

      // Возвращение идентификатора
      return(id);

   } // if
   else
      return(-1); // Проблема
} break;

case BLINKER_RUN: // Запуск алгоритма
{
   // Просмотр базы данных и обработка каждого огонька
   for (int index=0; index < 256; index++)
   {
      // Активен ли огонек?
      if (lights[index].state == -1)
      {
         // Обновление счетчика
```

```
          if (++lights[index].counter >=
             lights[index].off_time)
          {
             // Обнуление счетчика
             lights[index].counter = 0;
             // Изменение состояний
             lights[index].state =
                   -lights[index].state;

              // Обновление цвета
              lpddpal->SetEntries(0,
                 lights[index].color_index,
                 1,&lights[index].on_color);
           } // if
        } // if
        else if (lights[index].state == 1)
        {
          // Обновление счетчика
          if (++lights[index].counter >=
             lights[index].on_time)
          {
             // обнуление счетчика
             lights[index].counter = 0;

             // изменение состояний
             lights[index].state =
                -lights[index].state;

            // обновление цвета
             lpddpal->SetEntries(0,
                lights[index].color_index,
                1,&lights[index].off_color);
          } // if
        } // else if
      } // for index
    } break;

    default: break;
} // switch

// Успешное завершение
return(1);

} // Blink_Colors
```

НА ЗАМЕТКУ Полужирным шрифтом выделена часть кода, обновляющая записи палитры DirectDraw. Предполагается, что задан глобальный интерфейс палитры lpddpal.

Приведенная выше функция состоит из трех частей: инициализация, обновление и логика работы. При первом вызове функция инициализирует сама себя. Затем следующий фрагмент кода проверяет, команду какого вида должна выполнять функция — обновляющую или обработки палитры. Если запрошен обновляющий режим, выполняется

добавление, удаление или обновление мерцающих огней. Если же запрошен режим обработки, все огни преобразуются функцией в течение одного цикла работы. В этом режиме функцию можно применять после первого добавления одного или нескольких огоньков. Это делается путем задания общей структуры BLINKER с последующей ее передачей в функцию с помощью команды BLINKER_ADD; в результате функция возвращает идентификатор мерцающего огонька, который нужно сохранить. Этот идентификатор нужен для обновления или удаления созданного мерцающего огонька.

После того как созданы все нужные мерцающие огоньки, эту функцию можно вызывать для каждого кадра с параметром BLINKER_RUN. Пусть, например, игра работает с частотой 30 кадров/с и вы хотите, чтобы красный огонек секунду горел и на секунду отключался, а зеленый огонек — две секунды горел и на две секунды отключался. Кроме того, для красного и зеленого огоньков вы хотите использовать записи палитры с индексами 250 и 251 соответственно. Вот какой для этого нужен код:

```
BLINKER temp; // Хранилище временной информации

PALETTEENTRY red   = {255,0,0,PC_NOCOLLAPSE};
PALETTEENTRY green = {0,255,0,PC_NOCOLLAPSE};
PALETTEENTRY black = {0,0,0,PC_NOCOLLAPSE};

// Добавление красного огонька
temp.color_index = 250;
temp.on_color    = red;
temp.off_color   = black;
temp.on_time     = 30; // 30 циклов при частоте 30 кадров/с = 1 с
temp.off_time    = 30;

// Вызов функции
int red_id = Blink_Colors(BLINKER_ADD, &temp, 0);

// Создание зеленого огонька
temp.color_index = 251;
temp.on_color    = green;
temp.off_color   = black;
temp.on_time     = 60; // 60 циклов при частоте 30 кадров/с = 2 с
temp.off_time    = 60;
```

Все готово! Теперь в основном цикле игры можно вызывать функцию Blink_Colors(); это можно делать в течение каждого цикла примерно так:

```
// Вход в основной цикл игры
while(TRUE)
{
  // Проверка наличия в очереди сообщений,
  // извлечение сообщений при их наличии
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
  {
    // Проверка сообщения о завершении работы
    if (msg.message == WM_QUIT)
      break;

    // Пересчет всех ускоряющих ключей
    TranslateMessage(&msg);
```

```
    // Передача сообщения процедуре окна
    DispatchMessage(&msg);
  } // if

  // Основная логика игры
  Game_Main();

  // Переключение всех цветов;
  // эту часть лучше было бы поместить
  // в функцию Game_Main()
  Blink_Colors(BLINKER_RUN, NULL, 0);

} // while
```

Любой мерцающий огонек можно удалить, зная его идентификатор; после этого огонек станет недоступен для обработки. Например, красный огонек можно удалить с помощью такого кода:

```
Blink_Colors(BLINKER_DELETE, NULL, red_id);
```

Все очень просто. Кроме того, можно изменить значения интервалов времени, в течение которых огонек находится в выключенном и включенном состояниях, задав другую структуру BLINKER и вызвав функцию с параметром BLINKER_UPDATE. Например, вот код, предназначенный для изменения параметров зеленого мерцающего огонька:

```
// Установка новых параметров
temp.on_time  = 100;
temp.off_time  = 200;
temp.on_color  = {255,255,0,PC_NOCOLLAPSE};
temp.off_color = {0,0,0,PC_NOCOLLAPSE};

// Обновление мерцающего огонька
Blink_Colors(BLINKER_UPDATE, temp, green_id);
```

Чтобы увидеть все это "живьем", обратитесь к демонстрационной программе DEMO7_15.CPP на прилагаемом компакт-диске, в которой для создания мерцающих огоньков на космическом корабле используется функция Blink_Colors().

Циклическая перестановка цветов в 256-цветном режиме

Рассмотрим еще один интересный эффект цветовой анимации под названием *циклическая перестановка цветов* (color rotation) или *циклический сдвиг цветов* (color shifting). Суть его в том, что определенный набор цветов размещается в соседних регистрах, после чего эти цвета перемещаются в циклическом порядке (рис. 7.37). С помощью такого эффекта можно добиться, чтобы объекты казались движущимися; при этом не понадобится выводить на экран ни одного дополнительного пикселя. Это оказывается полезным, например, при имитации водной поверхности или движения потоков. Кроме того, можно вывести несколько изображений, задав для каждого из них свой цвет. При циклической перестановке этих цветов будет казаться, что изображения перемещаются.

Код, с помощью которого выполняется циклическая перестановка цветов, довольно тривиален. Чтобы осуществить такую перестановку в области, ограниченной элементами массива color[c1] и color[c2], можно воспользоваться таким кодом:

```
temp = color[c1];

for (int index = c1; index < c2; index++)
  color[c1] = color[c1+1];
```

```
// Завершение цикла
color[index] = temp;
```

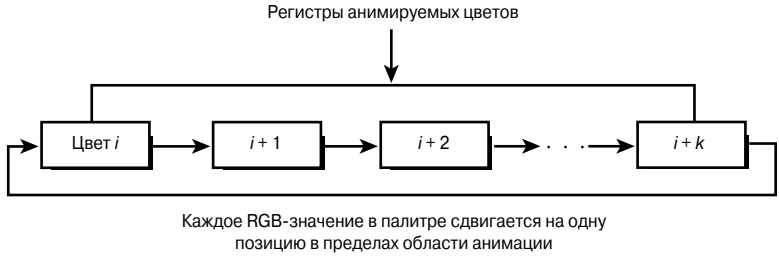


Рис. 7.37. Циклическая перестановка цветов

Ниже приводится функция, в которой реализуется используемый в нашей библиотеке алгоритм.

```
int Rotate_Colors(int start_index, int end_index)
{
// Функция осуществляет циклическую перестановку
// цветов в заданных пределах

int colors = end_index — start_index + 1;

PALETTEENTRY
     work_pal[MAX_COLORS_PALETTE]; // Рабочая палитра

// Извлечение палитры цветов
lpddpal->GetEntries(0,start_index,colors,work_pal);

// Сдвиг цветов
lpddpal->SetEntries(0,start_index+1,colors-1,work_pal);

// Перемещение последнего цвета
lpddpal->SetEntries(0,start_index,1,&work_pal[colors — 1]);

// Обновление "теневой"палитры
lpddpal->GetEntries(0,0,MAX_COLORS_PALETTE,palette);

// Успешное выполнение
return(1);

} // Rotate_Colors
```

В приведенную функцию передаются индексы цвета, ограничивающие область циклической перестановки, после чего реализуется сама перестановка. На “теневые палитры” не стоит обращать внимания, они относятся к библиотеке; сосредоточим внимание на логике. Чтобы показать работу этой функции, рассмотрим демонстрационную программу DEMO7_16.CPP, в которой с помощью функции создается поток кислоты (зануда скажет, что это вода).

Приемы, применяющиеся в режимах RGB

При работе в RGB-режимах невозможно применять трюки, связанные с цветами. Другими словами, каждый пиксель на экране характеризуется своими значениями в кодировке RGB, поэтому одно изменение не может повлиять на все изображение. Однако даже в этом случае есть два способа обработки данных:

- путем преобразования цветов или кодовой таблицы цветов вручную;
- с помощью новых подсистем DirectDraw, предназначенных для коррекции цветов и гамма-коррекции и выполняющих преобразование цветов первичной поверхности в реальном времени.

СЕКРЕТ

Термин *гамма-коррекция* (gamma correction) относится к нелинейному отклику монитора компьютера на входное воздействие. В большинстве случаев гамма-коррекция позволяет модифицировать интенсивность или яркость видеосигнала. Однако гамма-коррекцию можно выполнять для каждого из трех основных каналов (красного, зеленого и синего) независимо, получая при этом интересные эффекты.

Преобразование цветов вручную и таблицы цветов

Систему гамма-коррекции можно применять, как минимум, для операций фильтрации изображения как единого целого. Сначала рассмотрим таблицы цветов, применяемые в RGB-режимах, а затем обсудим входящую в DirectX систему гамма-коррекции.

Работая с пикселями, представленными значениями типа `WORD` в кодировке RGB, не удается найти никаких обходных путей для реализации цветовой анимации. Необходимо не только записать каждый пиксель, цвет которого нужно изменить, но, возможно, потребуется и считать их. В худшем случае понадобится выполнить чтение, преобразование и запись для каждого изменяемого пикселя. Другого пути просто нет.

Однако кое-что можно упростить. В большинстве случаев математические преобразования в RGB-пространстве требуют больших затрат вычислительных ресурсов. Пусть, например, нужно имитировать светящееся пятно прямоугольной формы с размерами (*width*, *height*) и вершиной в точке (x, y) в 16-битовом графическом режиме. Как это сделать?

Начнем со сканирования прямоугольной области, состоящей из пикселей, которые образуют светящееся пятно, и помещения их в битовый образ. Затем цвета каждого входящего в битовый образ пикселя подвергаются преобразованию примерно такого вида:

```
I*pixel(r,g,b) = pixel(I*r, I*g, I*b)
```

Интенсивность модулируется тремя операциями умножения. Кроме того, нужно сначала извлечь RGB-компоненты из 16-битовых значений типа `WORD`, а после преобразования поместить их обратно. Суть трюка в том, что при этом применяется кодовая таблица цветов.

Вместо того чтобы использовать все 65 536 цветов, доступных в 16-битовом режиме, выводятся объекты, которые могут быть раскрашены, скажем, 1024 цветами, равномерно распределенными по цветовому пространству объемом 64 Кбайт. После этого создается кодовая таблица цветов, содержащая двухмерный массив, объем которого кратен 1024. Кратность объема равна количеству интенсивностей цветов (например, 64). Затем берется RGB-уровень каждого реального цвета, вычисляется 64 его оттенка и заносятся в таблицу. После создания таким способом объекта появляется возможность использовать 16-битовое значение типа `WORD` в качестве одного индекса таблицы, а уровень яркости — в качестве другого ее индекса; записи в таблице представляют собой предвычисленные RGB-значения. Таким образом, операции с освещенностью сводятся к простому поиску.

Этот метод можно использовать для получения прозрачности, альфа-смешивания, осветления, затемнения и т.д. Демонстрационная программа будет показана только в следующей главе, но если вы хотите применить цветовые эффекты в 16-, 24- или 32-битовых режимах, то единственный способ для этого — применение таблиц цветов.

Новые интерфейсы DirectX

В DirectX 5.0 добавлены два новых интерфейса, которые предоставляют разработчикам игр и программистам, интенсивно работающим с видеоизображениями, возможность дополнительно контролировать цветовые свойства изображения на экране, не прибегая к сложным программным алгоритмам. Например, казалось бы, несложно добавить к изображению на экране немного красного цвета, изменить оттенок изображения и т.д. Однако подобные операции, которые в телевизоре сводятся к повороту ручки настройки, довольно сложно выполнить, преобразуя цифровые данные программными средствами. К счастью, два новых интерфейса, IDirectDrawGammaControl и IDirectDrawColorControl, позволяют выполнять эти изменения легко и просто.

Интерфейс IDirectDrawColorControl очень похож на интерфейс настройки телевизора. Он предоставляет контроль над яркостью, контрастом, тоном, насыщенностью, резкостью и общей контрастностью. Чтобы воспользоваться этим интерфейсом, необходимо запросить его в указателе первичной поверхности с помощью идентификатора IID_IDirectDrawColorControl. После извлечения интерфейса следует настроить приведенную ниже структуру DDCOLORCONTROL.

```
typedef struct _DDCOLORCONTROL
{
  DWORD dwSize;  // Размер структуры
  DWORD dwFlags; // Указывает корректные поля
  LONG  lBrightness;
  LONG  lContrast;
  LONG  lHue;
  LONG  lSaturation;
  LONG  lSharpness;
  LONG  lGamma;
  LONG  lColorEnable;
  DWORD dwReserved1;
} DDCOLORCONTROL, FAR *LPDDCOLORCONTROL;
```

Затем нужно вызвать функцию IDIRECTDRAWCOLORCONTROL::SetColorControl(), после выполнения которой первичная поверхность будет тут же модифицирована. Воздействие этих эффектов будет продолжаться до следующего вызова функции.

```
HRESULT SetColorControl(LPDDCOLORCONTROL lpColorControl);
```

Принцип управления гамма-коррекцией несколько иной. Вместо “телевизороподобных” настроек программисту предоставляется возможность управлять линейно изменяющимися цветовыми шаблонами первичной поверхности для красного, зеленого и синего цветов. Настройка интерфейса IDirectDrawGammaControl аналогична настройке интерфейса управления цветами, поэтому я не стану излагать этот вопрос (как будто я сумел хорошо объяснить, как управлять цветами...). Более подробную информацию можно найти в документации DirectX SDK. Полученные там сведения позволят вам легко имитировать вспышки, сверкание молний, наступление темноты, подводные сцены и т.п. Единственная проблема состоит в том, что все это будет работать только при наличии соответствующей аппаратной под-

держки, которая встречается на видеокартах крайне редко. У меня нет ни одной такой карты, поэтому на сей раз придется обойтись без демонстрационного примера.

Совместное использование GDI и DirectX

Интерфейс графического устройства GDI (Graphics Device Interface) — это Win32-подсистема вывода изображений в Windows. В предыдущих разделах, посвященных программированию в Windows, вы уже познакомились с ее работой, и я не стану еще раз рассказывать о контексте устройства и тому подобных вещах.

Все, что нужно для совместного использования GDI и DirectDraw, — запросить у DirectDraw совместимый DC (device context — контекст устройства), а затем использовать его точно так же, как будто он получен стандартным вызовом функции GetDC(). Использование совместимого контекста устройства ничем не отличается от использования контекста устройства, полученного стандартным методом.

Может возникнуть вопрос, как GDI будет работать с DirectDraw в ситуации, когда DirectDraw принимает на себя полное управление графической системой (как это происходит в полноэкранном режиме)? На самом деле Windows не может использовать GDI для вывода в поверхности DirectDraw при работе в исключительном режиме, но это может делать *программист*. Windows обменивается сообщениями со своими подсистемами, такими, как GDI, MCI и др. Когда DirectDraw получает исключительный контроль над аппаратными системами, сообщения не обрабатываются. Например, вызов графической функции GDI, предназначенной для вывода какого-нибудь изображения, в полноэкранном режиме отбрасывается.

Однако программист может в любой момент использовать программные средства перечисленных подсистем. Поскольку вся графика выводится в поверхности, логично предположить, что есть возможность запросить GDI-совместимый интерфейс из поверхности DirectDraw. Так оно и есть. Функция, с помощью которой можно выполнить это действие, носит имя IDIRECTDRAWSURFACE7::GetDC(), а ее прототип имеет такой вид:

```
HRESULT GetDC(HDC FAR *lphDC);
```

Все, что нужно сделать, — вызвать эту функцию, указав место хранения DC, например:

```
LPDIRECTDRAWSURFACE7 lpdds; // Считаем, что это корректный указатель

HDC xdc; // DC DirectX

if (FAILED(lpdds->GetDC(&xdc)))
  { /* ошибка */ }

// Действия, которые нужно выполнить с DC...
```

Завершив работу с DirectDraw-совместимым DC, его необходимо освободить точно так же, как и обычный контекст устройства. Предназначенная для этого функция IDIRECTDRAWSURFACE7::ReleaseDC() приведена ниже.

```
HRESULT ReleaseDC(HDC hDC);
```

В эту функцию нужно просто передать запрошенный перед этим DC:

```
if (FAILED(lpdds->ReleaseDC(xdc)))
  { /* ошибка */ }
```

ВНИМАНИЕ

Если поверхность заблокирована, функция GetDC() работать не будет, так как она тоже блокирует поверхность. Кроме того, после извлечения контекста устройства из поверхности не забудьте отключить его, как только с ним будут выполнены все необходимые действия, потому что функция GetDC() создает внутреннюю блокировку поверхности и к ней не будет доступа. По сути, записывать данные в поверхность можно либо с помощью GDI, либо с помощью DirectDraw, но не с помощью обоих интерфейсов одновременно.

В качестве примера применения GDI рассмотрим демонстрационную программу DEMO7_17.CPP. В ней создается полноэкранное приложение DirectDraw в режиме 640×480×256, а затем с помощью GDI в случайном месте экрана выводится текстовый фрагмент. Ниже приводится код, с помощью которого осуществляется этот вывод.

```
int Draw_Text_GDI(char *text, int x,int y,
          COLORREF color,
          LPDIRECTDRAWSURFACE7 lpdds)
{
// Функция выводит указанный текст на заданную поверхность;
// применяя цветовой индекс в качестве цвета палитры

HDC xdc; // Рабочий dc

// Извлечение dc из поверхности
if (FAILED(lpdds->GetDC(&xdc)))
  return(0);

// Настройка цвета текста
SetTextColor(xdc,color);

// Установка прозрачного фонового режима;
// область черного цвета не копируется
SetBkMode(xdc, TRANSPARENT);

// Вывод текста
TextOut(xdc,x,y,text,strlen(text));

// Освобождение dc
lpdds->ReleaseDC(xdc);

// Успешное завершение
return(1);
} // Draw_Text_GDI
```

СЕКРЕТ

Обратите внимание, что цвет задается как значение типа COLORREF. Это очень важный момент. Значения этого типа представляют собой 24-битовые структуры в кодировке RGB. При этом возникает проблема: при работе DirectX в режиме с палитрой необходимо подобрать цвет, ближайший к запрошенному. На фоне медленной работы GDI это приводит к очень медленному выводу текста с помощью этого интерфейса. Если программа интенсивно выдает текстовые сообщения, настоятельно рекомендую написать собственный блиттер текстов.

Все действия с DC выполняются в теле функции, поэтому все, что нам нужно, — это ее вызвать. Например, чтобы вывести на первичную поверхность сообщение “Всем привет!” чисто зеленого цвета, которое отобразится в точке с координатами (100, 100), следует воспользоваться строкой

```
Draw_Text_GDI("Всем привет!", 100,100,
        RGB(0,255,0), lpddsprimary);
```

Прежде чем продолжить, я хочу сказать несколько слов о ситуациях, в которых следует применять интерфейс GDI. Вообще говоря, GDI работает медленно, и обычно я использую его для вывода текста, графики и прочего только на этапе разработки. Кроме того, на этом этапе он оказывается очень полезным для медленной эмуляции. Предположим, вы собираетесь написать высокопроизводительную функцию Draw_Line(), предназначенную для вывода на экран текстовых строк, но еще не успели это сделать; в такой ситуации данную функцию всегда можно эмулировать с помощью GDI. Таким образом вы сможете вывести на экран все, что нужно, а позже написать для этих целей более производительный код.

Вглубь DirectDraw

Как вы убедились, DirectDraw — довольно сложная графическая система. Она содержит ряд интерфейсов, в каждом из которых множество функций. Основная особенность технологии DirectDraw — единый подход к использованию аппаратного обеспечения. Затраченные на изучение DirectDraw усилия разработчика игр окупаются, позволяя использовать свойства и/или возможности различных интерфейсов DirectDraw и выполнять нужные действия в зависимости от возможностей системы. Список сведений, которые могут понадобиться при настройке игры в системе, очень велик. Получить всю необходимую информацию о видеосистеме позволяют функции GetCaps() или Get*(), которые содержатся во всех основных интерфейсах. Рассмотрим наиболее полезные функции GetCaps().

Основной объект DirectDraw

Сам по себе объект DirectDraw представляет видеокарту и описывает HEL (уровень эмуляции аппаратных средств) и HAL (уровень аппаратных абстракций). Интересующая нас функция называется IDIRECTDRAW7::GetCaps().

```
HRESULT GetCaps(
  LPDDCAPS lpDDDriverCaps, // Указатель на свойства HAL
  LPDDCAPS lpDDHELCaps);  // Указатель на свойства HEL
```

Эту функцию можно использовать как для запроса свойств HEL, так и для запроса свойств HAL, получая интересующую вас информацию в структуре DDCAPS. Вот пример такого кода:

```
DDCAPS hel_caps, hal_caps;

// Инициализация структур
DDRAW_INIT_STRUCT(hel_caps);
DDRAW_INIT_STRUCT(hal_caps);

// Вызов функции
if (FAILED(lpdd->GetCaps(&hal_caps, &hel_caps)))
  return(0);
```

После этого у вас есть полная информация об обоих уровнях. Вот как выглядит структура DDCAPS:

```
typedef struct _DDCAPS
{
  DWORD dwSize;
  DWORD dwCaps;    // Возможности, связанные с драйвером
```

```
DWORD dwCaps2;         // Дополнительные возможности,
                       // связанные с драйвером
DWORD dwCKeyCaps;    // Параметры цветовых ключей
DWORD dwFXCaps;        // Возможности расширения и эффектов
DWORD dwFXAlphaCaps; // Альфа-возможности
DWORD dwPalCaps;       // Возможности палитры
DWORD dwSVCaps;        // Возможности стереоотображения
DWORD dwAlphaBltConstBitDepths;          // Битовая глубина альфа-канала
DWORD dwAlphaBltPixelBitDepths;          // .
DWORD dwAlphaBltSurfaceBitDepths;        // .
DWORD dwAlphaOverlayConstBitDepths;     // .
DWORD dwAlphaOverlayPixelBitDepths;      // .
DWORD dwAlphaOverlaySurfaceBitDepths;  // .
DWORD dwZBufferBitDepths;       // Битовая глубина Z-буфера
DWORD dwVidMemTotal;             // Общий объем видеопамяти
DWORD dwVidMemFree;              // Общий объем свободной видеопамяти
DWORD dwMaxVisibleOverlays;     // Максимальное количество видимых наложений
DWORD dwCurrVisibleOverlays;     // Наложения, видимые в данный момент
DWORD dwNumFourCCCodes;         // Количество поддерживаемых кодов FOURCC
DWORD dwAlignBoundarySrc;       // Ограничения на выравнивание наложений
DWORD dwAlignSizeSrc;            // .
DWORD dwAlignBoundaryDest;      // .
DWORD dwAlignSizeDest;           // .
DWORD dwAlignStrideAlign;        // Выравнивание шага
DWORD dwRops[DD_ROP_SPACE]; // Поддерживаемые растровые параметры
DWORD dwReservedCaps;            // Зарезервированное поле
DWORD dwMinOverlayStretch;       // Коэффициенты растяжения наложения
DWORD dwMaxOverlayStretch;       // .
DWORD dwMinLiveVideoStretch;    // Поле, вышедшее из употребления
DWORD dwMaxLiveVideoStretch;    // .
DWORD dwMinHwCodecStretch;      // .
DWORD dwMinHwCodecStretch;      // .
DWORD dwReserved1;   // Зарезервированное поле
DWORD dwReserved2;   // .
DWORD dwReserved3;   // .
DWORD dwSVBCaps;     // Возможности, связанные с блиттингом из системы в видеоизображение
DWORD dwSVBCKeyCaps; // .
DWORD dwSVBFXCaps;     // .
DWORD dwSVBRops[DD_ROP_SPACE]; // .
DWORD dwVSBCaps;       // Возможности, связанные с блиттингом из видеоизображения в систему
DWORD dwVSBCKeyCaps;     // .
DWORD dwVSBFXCaps;       // .
DWORD dwVSBRops[DD_ROP_SPACE]; // .
DWORD dwSSBCaps; // Возможности, связанные с блиттингом из системной памяти в
                 // системную память
DWORD dwSSBCKeyCaps;    // .
DWORD dwSSBFXCaps;       // .
DWORD dwSSBRops[DD_ROP_SPACE]; // .
DWORD dwMaxVideoPorts; // Максимальное количество портов в реальном времени
DWORD dwCurrVideoPorts; // Текущее количество портов в реальном времени
DWORD dwSVBCaps2; // Дополнительные возможности, связанные с блиттингом из системы
                  // в видеоизображение
DWORD dwNLVBCaps; // Возможности блиттинга из нелокальной
```

```
                                // видеопамяти в локальную
  DWORD dwNLVBCaps2;        // .
  DWORD dwNLVBCKeyCaps;     // .
  DWORD dwNLVBFXCaps;       // .
  DWORD dwNLVBRops[DD_ROP_SPACE]; // .
  DDSCAPS2 ddsCaps          // Общие возможности поверхности
  } DDSCAPS, FAR* LPDDCAPS;
```

Для того чтобы описать все поля приведенной выше структуры, потребовалась бы отдельная книга, поэтому ищите эту информацию в документации SDK. Смысл большинства полей понятен из их названия. Например, DDCAPS.dwVidMemFree — один из моих любимых членов структуры, потому что он указывает на количество видеопамяти, доступной для использования поверхностями.

Есть еще одна неплохая функция под названием GetDisplayMode(). С ее помощью можно получить информацию о видеорежиме, в котором работает система в оконном режиме. Ниже приведен прототип этой функции.

```
HRESULT GetDisplayMode(LPDDSURFACEDESC2 lpDDSurfaceDesc2);
```

Структура DDSURFACEDESC2 нам уже встречалась, поэтому применение функции GetDisplayMode() не должно вызвать у вас затруднений.

Получение информации о поверхностях

В большинстве случаев о запросе свойств поверхности можно не беспокоиться, потому что вы их уже знаете (поскольку сами их создали). Тем не менее свойства первичной поверхности и поверхности заднего буфера являются очень важными, так как помогают проникнуть в суть аппаратных свойств каждой из них. Общие свойства поверхности можно узнать с помощью функции IDIRECTDRAWSURFACE7::GetCaps(). Прототип этой функции имеет такой вид:

```
HRESULT GetCaps(LPDDSCAPS2 lpDDSCaps);
```

Функция GetCaps() возвращает стандартную структуру DDSCAPS2, которая нам уже знакома. После вызова функции останется извлечь всю нужную информацию из этой структуры.

Следующая интересующая нас функция, которая имеет отношение к поверхностям, называется IDIRECTDRAWSURFACE7::GetSurfaceDesc(). Эта функция заполняет информацией структуру DDSURFACEDESC2, в которой содержится несколько более подробная информация о самой поверхности. Ниже приведен прототип функции GetSurfaceDesc().

```
HRESULT GetSurfaceDesc(LPDDSURFACEDESC2 lpDDSurfaceDesc);
```

Существует также метод IDIRECTDRAWSURFACE7::GetPixelFormat(), однако о нем уже шла речь ранее. Кроме того, функция GetSurfaceDesc() также возвращает формат пикселя в поле DDSURFACEDESC2.ddpfPixelFormat.

Получение информации о палитрах

О палитрах много не расскажешь. DirectDraw предоставляет лишь одно значение типа WORD в битовой кодировке, описывающее особенности любой палитры. Возвращающая это значение функция называется IDIRECTDRAWPALETTE::GetCaps(), а ее прототип приведен ниже.

```
HRESULT GetCaps(LPDWORD lpdwCaps);
```

lpdwCaps — это указатель на значение типа WORD; список возможных флагов приведен в табл. 7.7.

Таблица 7.7. Флаги, описывающие свойства палитры

Значение	*Описание*
DDPCAPS_1BIT	Поддерживает 1-битовые палитры
DDPCAPS_2BIT	Поддерживает 2-битовые палитры
DDPCAPS_4BIT	Поддерживает 4-битовые палитры
DDPCAPS_8BIT	Поддерживает 8-битовые палитры
DDPCAPS_8BITENTRIES	Используется индексная палитра
DDPCAPS_ALPHA	В каждой записи палитры поддерживается альфа-компонент
DDPCAPS_ALLOW256	Палитра позволяет задавать все 256 цветов
DDPCAPS_PRIMARYSURFACE	Палитра присоединена к первичной поверхности
DDPCAPS_VSYNC	Палитру можно модифицировать синхронно с обновлением монитора

DirectDraw и оконные режимы

Последняя тема, которую хотелось бы раскрыть в этой главе, — это применение DirectDraw в оконном режиме. При запуске игры в оконном режиме возникает проблема, связанная с тем, что разработчику предоставляется крайне мало возможностей по управлению глубиной цвета и разрешением. Писать приложения DirectDraw, которые запускаются в полноэкранном режиме, довольно сложно, однако в оконном режиме работать намного труднее. Следует принимать во внимание, что пользователь может запускать приложение при любом разрешении и в любом цветовом режиме, что может сказаться на производительности данного приложения. Однако в оконном режиме возникает не только эта проблема. Например, игра может быть разработана только для 8- или 16-битового режима. В этом случае возможен сбой работы программы, если пользователь запустит ее в режиме с большей глубиной цвета, чем предусмотрено.

Хотя создание игр, работающих и в оконном и в полноэкранном режимах, — самый лучший вариант, в большинстве случаев я собираюсь придерживаться полноэкранного режима, чтобы упростить разработку демонстрационных примеров. Однако наряду с этим я создал несколько оконных приложений, работающих, в частности, при разрешении 800×600 или более высоком с глубиной цветов, равной 8 бит. Это облегчает отладку приложений DirectX; кроме того, такие режимы позволяют организовать дополнительные (в том числе графические) окна вывода. Рассмотрим, как создаются оконные приложения DirectX и как осуществляется управление первичной поверхностью.

Что касается оконных приложений DirectDraw, то необходимо помнить следующее: первичная поверхность — это вся площадь экрана, а не только окно! Рис. 7.38 служит графической иллюстрацией этого утверждения, которое означает, что выводить изображение на экран дисплея вслепую нельзя, иначе вы рискуете вторгнуться в клиентские области окон других приложений. Если создается программа сохранения экрана или какая-либо другая программа, предназначенная для управления экраном, то это может входить в планы разработчика, но в большинстве случаев в его намерения входит вывод только в клиентскую область окна приложения. Это означает, что программист каким-то образом должен определить координаты клиентского окна с тем, чтобы вывод осуществлялся только в эту область.

Вторая проблема, возникающая при разработке оконных приложений, связана с отсечением. Если вы намерены использовать функцию Blt(), то учтите, что она ничего не

знает о клиентском окне и будет выводить изображение за его рамки. Это означает, что каким-то образом нужно сообщить системе отсечения DirectDraw о наличии на экране окна. Система отсечения должна ограничивать все изображения рамками окна, независимо от размеров этого окна или его положения на экране.

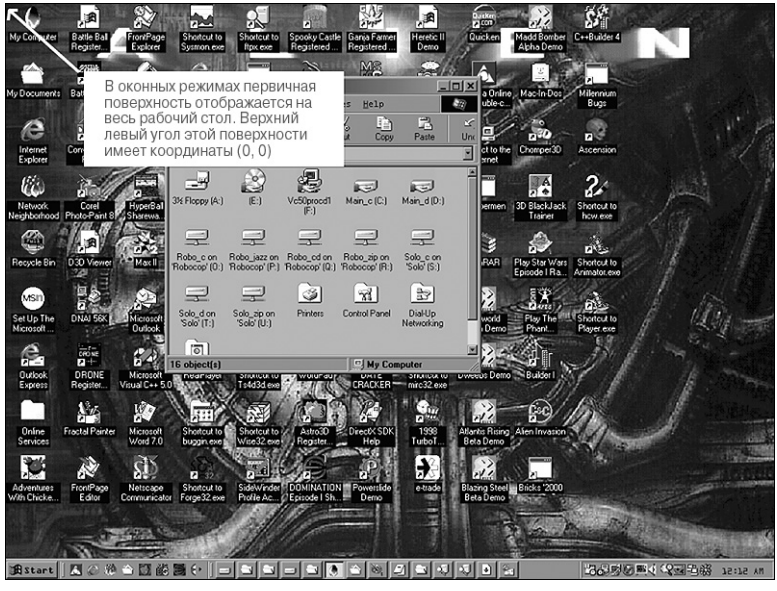


Рис. 7.38. В оконном режиме в первичную поверхность DirectDraw отображается весь рабочий стол

При этом возникает еще одна проблема: что будет происходить при перемещении или изменении размера окна пользователем? Конечно же, размер окна можно зафиксировать, однако перемещение должно работать в любом случае, иначе просто нет смысла создавать данное приложение в оконном режиме. Для решения этой проблемы необходимо научиться отслеживать сообщения `WM_SIZE` и `WM_MOVE`.

Далее существует проблема, связанная с 8-битовыми палитрами. Если установлен 8-битовый режим, а вам нужно изменить палитру, возникают трудности. В Windows есть диспетчер палитры (Palette Manager), о наличии которого следует помнить при внесении изменений в 8-битовую палитру, потому что это действие может значительно ухудшить внешний вид других приложений Windows. В большинстве случаев палитру можно изменять безболезненно, однако около 20 ее записей лучше оставить в покое (системные цвета и цвета Windows). В этом случае диспетчер палитры может позаботиться о том, чтобы другие приложения выглядели так, как они должны выглядеть.

Наконец, наиболее очевидные проблемы — это те, которые связаны с блиттингом битов и манипуляцией пикселями в различных режимах. Таким образом, чтобы создать надежное приложение, нужно написать код для его работы со всеми возможными глубинами цветов.

Итак, приступим к рассмотрению особенностей оконных режимов. Вы уже знаете, как создавать оконные приложения. Единственное, о чем следует помнить, — это то, что в оконном режиме не задается видеорежим и не создается вторичный задний буфер. В этом режиме нельзя выполнять переключение страниц. Чтобы организовать двойной буфер, необходимо либо воспользоваться блиттером, либо решать эту задачу вручную. Однако создать сложную цепочку поверхностей, содержащую последовательность изображений, а затем вызвать функцию `Flip()` нельзя. Этот метод не работает. На самом деле это

не проблема — в такой ситуации создается другая поверхность, размер которой совпадает с размером клиентской области окна; в этой поверхности формируется изображение, после чего осуществляется блиттинг в клиентскую область окна, которая находится в первичном буфере. Таким путем удается избежать мерцания экрана.

Ниже приведен код, предназначенный для создания оконного приложения DirectDraw. Сначала выполняется инициализация DirectDraw.

```
LPDIRECTDRAW7 lpdd = NULL; // Интерфейс IDirectDraw7

// Создание интерфейса IDirectDraw7
if (FAILED(DirectDrawCreateEx(NULL, (void**)&lpdd,
      IID_IDirectDraw7, NULL)))
  return(0);


// Установка полноэкранного уровня взаимодействия
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle,
                  DDSCL_NORMAL)))
  return(0);

// Обнуление ddsd и установка размера
DDRAW_INIT_STRUCT(ddsd);

// Установка полей
ddsd.dwFlags = DDSD_CAPS;

// Запрос первичной поверхности
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Создание первичной поверхности
if (FAILED(lpdd->CreateSurface(&ddsd,&lpddsprimary,NULL)))
  return(0);
```

Основная особенность приведенного выше кода — установка уровня взаимодействия. Обратите внимание, что в соответствующем поле функции SetCooperativeLevel() вместо обычной комбинации флагов (DDSCL_FULLSCREEN|DDSCL_ALLOWMODEX|DDSCL_EXCLUSIVE|DDSCL_ALLOWREBOOT), применяющейся в полноэкранных режимах, фигурирует флаг DDSCL_NORMAL.

Кроме того, во время создания окна приложения вместо флага WS_POPUP вы, возможно, предпочтете использовать флаг WS_OVERLAPPED или WS_OVERLAPPEDWINDOW. При использовании стиля WS_POPUP окно создается без заголовка, без элементов управления и т.д. Флаг WS_OVERLAPPED создает окно с заголовком, однако такое, размеры которого не поддаются изменению. Если применить стиль WS_OVERLAPPEDWINDOW, мы получим полнофункциональное окно со всеми управляющими элементами. Однако в большинстве случаев я предпочитаю флаг WS_OVERLAPPED, чтобы не связываться с проблемами изменения размеров.

В качестве примера, в котором реализованы все полученные до этого момента знания, рассмотрим демонстрационную программу DEMO7_18.CPP. В ней создается оконное приложение DirectX с клиентской областью 400×400 и первичная поверхность.

Вывод пикселей в окне

Перейдем к рассмотрению того, как получить доступ к клиентской области окна. При этом следует помнить, что первичная поверхность совпадает со всем экраном и глубина цвета заранее не известна. Сначала рассмотрим первую проблему — нахождение геометрических параметров клиентской области окна.

Поскольку пользователь может перемещать окно в любое место экрана, координаты клиентской области относительно левого верхнего угла экрана (его координаты равны (0,0)) изменяются. Нужно найти способ определять координаты верхнего левого угла клиентской области относительно начала отсчета. Затем полученные координаты используются в качестве начала отсчета при выводе пикселей. Для этого понадобится функция GetWindowRect() (которая уже упоминалась ранее).

```
BOOL GetWindowRect(HWND hWnd, // Дескриптор окна
        LPRECT lpRect);       // Адрес структуры, предназначенной для координат окна
```

ВНИМАНИЕ

Фактически функция GetWindowRect() запрашивает координаты всего окна, включая его управляющие элементы и рамку. Позже я покажу вам, как находить точные координаты клиентской области.

Если в функцию передать дескриптор окна приложения, она возвратит экранные координаты клиентской области, которые будут храниться в переменной lpRect. Таким образом, нужно лишь запросить экранные координаты верхнего левого угла окна, вызвав функцию GetWindowRect(). Конечно же, при каждом перемещении окна эти координаты изменяются, поэтому рассматриваемую функцию необходимо вызывать в каждом кадре или после получения сообщения WM_MOVE. Я предпочитаю вызывать функцию GetWindowRect() для каждого кадра; в этом случае нет необходимости обрабатывать больше сообщений Windows, чем нужно!

Получив в свое распоряжение экранные координаты клиентской области окна, можно приступать к работе с пикселями. Но ведь нам пока неизвестен формат пикселей! Как же его узнать?

Я рад, что вы об этом спросили, потому что знаю ответ на этот вопрос. В начале программы необходимо использовать функцию GetPixelFormat(), чтобы определить глубину цвета. Впоследствии полученная информация понадобится для вызовов различных функций, предназначенных для вывода пикселей. Таким образом, где-то в программе (возможно, в теле функции Game_Init()) после настройки DirectDraw следует вставить вызов функции GetPixelFormat(). Это можно сделать, например, так:

```
int pixel_format = 0; // Глобальная переменная для хранения
                      // количества байтов в пикселе
DDPIXELFORMAT ddpixelformat; // Формата пикселя

// Обнуление и настройка структуры
DDRAW_INIT_STRUCT(ddpixelformat);

// Запрос формата пикселей
lpddsprimary->GetPixelFormat(&ddpixelformat);

// Настройка глобального формата пикселей
pixel_format = ddpixelformat.dwRGBBitCount;
```

Зная формат пикселей, его можно использовать в различных условных операциях, указателях функций или виртуальных функциях, чтобы задавать правильную глубину цветов в функциях, предназначенных для вывода пикселей. Проще всего воспользоваться условными операторами, проверяющими перед выводом изображений значение глобальной переменной pixel_format. С помощью приведенного ниже кода в клиентскую область окна выводится пиксель случайного цвета со случайными координатами.

```
DDSURFACEDESC2 ddsd;  // Описание поверхности DirectDraw
RECT       client;    // Хранение клиентского прямоугольника
```

```
// Запрос экранных координат клиентского прямоугольника окна
GetWindowRect(main_window_handle, &client);

// Инициализация структуры
DDRAW_INIT_STRUCT(ddsd);

// Блокировка первичной поверхности
lpddsprimary->Lock(NULL,&ddsd,
        DDLOCK_SURFACEMEMORYPTR| DDLOCK_WAIT,NULL);

// Запрос указателя первичной поверхности
// Преобразование к типу UCHAR*
UCHAR *primary_buffer = (UCHAR*)ddsd.lpSurface;

// Зависимость от глубины цветов
if (pixel_format == 32)
{
   // Вывод 10 случайных пикселей в 32-битовом режиме
   for (int index=0; index<10; index++)
   {
      int x = rand()%(client.right — client.left)
            + client.left;
      int y = rand()%(client.bottom — client.top)
            + client.top;
      DWORD color =_RGB32BIT(0,rand()%256,rand()%256,
                  rand()%256);
      *((DWORD*)(primary_buffer + x*4 + y*ddsd.lPitch)) =
         color;
   } // if 24 bit
   else if (pixel_format == 24)
   {
   // Вывод 10 случайных пикселей в 24-битовом режиме
   for (int index=0; index<10; index++)
   {
      int x = rand()%(client.right — client.left) +
            client.left;
      int y = rand()%(client.bottom — client.top) +
            client.top;
      ((primary_buffer + x*3 + y*ddsd.lPitch))[0] =
            rand()%256;
      ((primary_buffer + x*3 + y*ddsd.lPitch))[1] =
            rand()%256;
      ((primary_buffer + x*3 + y*ddsd.lPitch))[2] =
            rand()%256;
   } // for index
} // if 24 bit
else if (pixel_format == 16)
{
   // Вывод 10 случайных пикселей в 16-битовом режиме
   for (int index=0; index<10; index++)
   {
      int x = rand()%(client.right — client.left) +
            client.left;
```

```
    int y = rand()%(client.bottom — client.top) +
        client.top;
    USHORT color = _RGB16BIT565(rand()%256, rand()%256,
                    rand()%256);
    *((USHORT*)(primary_buffer + x*2 + y*ddsd.lPitch))
        = color;
  } // for index
} // if 16 bit
else
{  // Считаем, что в пикселе 8 бит
  // Вывод 10 случайных пикселей в 8-битовом режиме
  for (int index=0; index<10; index++)
  {
    int x = rand()%(client.right — client.left) +
        client.left;
    int y = rand()%(client.bottom — client.top) +
        client.top;
    UCHAR color = rand()%256;
    primary_buffer[x + y*ddsd.lPitch] = color;
} // for index
} // else

// Снятие блокировки первичного буфера
lpddsprimary->Unlock(NULL);
```

СЕКРЕТ

Оптимизация приведенного выше кода, несомненно, находится в кошмарном состоянии. Мне даже неловко показывать такой медленный, топорный и нелепый код, однако его преимущество в том, что он понятен. На самом деле лучше было бы применить указатели функций или виртуальную функцию, полностью избавившись от всех операций умножения и вычисления остатка от деления, а также воспользовавшись инкрементной адресацией.

В качестве примера вывода пикселей, которые могут иметь любую глубину цветов, рассмотрим программу DEMO7_19.CPP. В этой программе создается окно размером 400×400, а затем в клиентскую область выводятся пиксели (ну, почти только в клиентскую). Попробуйте запустить эту программу в режимах с различной глубиной цветов; при этом она должна работать во всех режимах! После окончания тестирования программы возвращайтесь к чтению книги, и я подробнее расскажу вам о внутренней клиентской области...

Определение параметров реальной клиентской области

При создании окна с помощью функции CreateWindow() или CreateWindowEx() возникает проблема, связанная с тем, что параметрами этих функций являются полная ширина и высота окна, включая его управляющие элементы. Таким образом, если создается окно в стиле WS_POPUP, в котором элементы управления отсутствуют, размер окна точно совпадает с размером клиентской области. Когда же к окну добавляются управляющие элементы, меню, границы и т.п., а также вызывается функция CreateWindowEx(), клиентская область окна сужается, чтобы все это могло уместиться. Если добавление осуществляется за счет рабочей области окна, то в результате она оказывается меньшей, чем нужно. Эта дилемма космического масштаба проиллюстрирована на рис. 7.39. Ее решение заключается в том, чтобы увеличить размеры окна, добавив место для его границы, управляющих элементов и т.п.

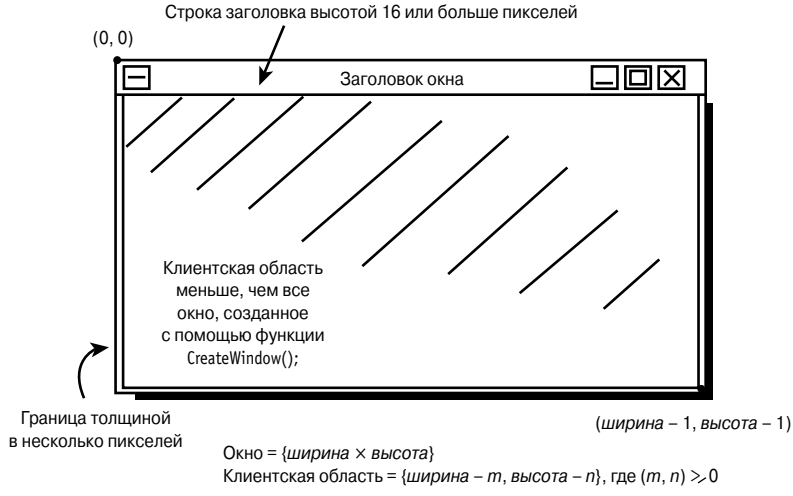


Рис. 7.39. Клиентская область окна меньше, чем все окно

Предположим, нам нужно создать окно, рабочая область которого имеет размеры 640×480, с границей, меню и стандартными элементами управления. Для этого нужно подсчитать, сколько дополнительных пикселей займут все эти дополнительные элементы в горизонтальном и вертикальном направлениях, а затем соответствующим образом увеличить размеры окна. В нашем распоряжении имеется замечательная функция AdjustWindowRectEx(), вычисляющая размер окна в различных стилях. Ниже приводится прототип этой функции.

```
BOOL AdjustWindowRectEx(
  LPRECT lpRect,      // Указатель на структуру с
                      // параметрами клиентской области
  DWORD dwStyle,      // Стили окна
  BOOL  bMenu,        // Флаг наличия меню
  DWORD dwExStyle); // Расширенный стиль
```

Достаточно передать все параметры в функцию — и она выполнит преобразование данных структуры, переданной через указатель lpRect. Здесь принимаются во внимание все дополнительные стили и флаги окна. Чтобы воспользоваться функцией AdjustWindowRectEx(), сначала нужно задать структуру RECT, указав в ней необходимый размер клиентской области, скажем, 640×480. После этого вызывается функция, в которую передаются все надлежащие параметры, с которыми было создано исходное окно. Однако, если программисту трудно держать в памяти параметры окна и имена флагов, он может запросить эту информацию у Windows. Ниже приведен код, в котором вызывается функция AdjustWindowRectEx(), а также вспомогательные функции Windows, запрашивающие стили на основе дескриптора HWND.

```
// Размер клиентской области
RECT window_rect = {0,0,640,480};

// Вычисление размеров окна
AdjustWindowRectEx(&window_rect,
  GetWindowStyle(main_window_handle),
  GetMenu(main_window_handle) != NULL,
  GetWindowExStyle(main_window_handle));

// Изменение размеров окна с помощью функции MoveWindow()
MoveWindow(main_window_handle,
```

```
CW_USEDEFAULT, // Положение по горизонтали
CW_USEDEFAULT, // Положение по вертикали
window_rect.right — window_rect.left,    // Ширина
window_rect.bottom — window_rect.top, // Высота
FALSE);
```

Вот и все! Если бы создавалась библиотека, приведенный выше код играл бы в ней крайне важную роль. Дело в том, что, если запрашивается окно определенного размера, подразумевается, что такую величину будет иметь клиентская область. Таким образом, если окно содержит какие-нибудь управляющие элементы, то его размеры ***ДОЛЖНЫ*** быть изменены.

Отсечение в окнах DirectX

Следующая задача — организовать отсечение в окне. Необходимо усвоить, что отсечение играет роль только для блиттера; оно не оказывает влияния на операции, непосредственно выполняемые с первичной поверхностью, поскольку первичная поверхность — это фактически ***весь*** экран, т.е. все то, что выводится на рабочий стол.

Интерфейс IDIRECTDRAWCLIPPER вам уже знаком, и я не стану к нему возвращаться. (Правда, я сам еще не уверен, что понимаю природу координат, фигурирующих в структуре RECT.) Прежде всего нужно создать отсекатель DirectDraw. Это делается так:

```
LPDIRECTDRAWCLIPPER lpddclipper // Переменная для
        = NULL;                 // хранения отсекателя

if (FAILED(lpdd->CreateClipper(0,&lpddclipper,NULL)))
  return(0);
```

Затем необходимо присоединить этот отсекатель к окну приложения с помощью функции IDIRECTDRAWCLIPPER::SetHWnd(). При этом отсекатель связывается с окном; кроме того, эта функция помогает выполнять все действия, связанные с изменением размеров и положения окна. Фактически программисту не нужно даже передавать список отсечения — все происходит автоматически. Эта функция очень простая. Ниже приведен ее прототип.

```
HRESULT SetHWnd(DWORD dwFlags, // Не используется; задаем 0
        HWND hWnd);            // Дескриптор окна
```

Присоединить отсекатель к основному окну можно с помощью такого вызова функции:

```
if (FAILED(lpddclipper->SetHWnd(0, main_window_handle)))
  return(0);
```

Затем этот отсекатель нужно связать с поверхностью, в которой будет выполняться отсечение (в данном случае с первичной поверхностью). Для этого применяется уже знакомая нам функция SetClipper():

```
if (FAILED(lpddsprimary->SetClipper(lpddclipper)))
  return(0);
```

ВНИМАНИЕ

Все готово, но есть одна проблема, связанная с DirectX. Значение счетчика ссылок на отсекатель равно 2, — оно увеличилось на 1 в момент создания отсекателя и еще на 1 при вызове функции SetClipper(). В этой ситуации удаление поверхности не уничтожит отсекатель. Чтобы от него избавиться, после удаления поверхности с помощью вызова функции lpddsprimary->Release() придется еще раз воспользоваться этой функцией: lpddclipper->Release(). Неопытный программист может попасть в ловушку, подумав, что удалил отсекатель с помощью инструкции lpddclipper->Release(), но это снижает показания счетчика только на единицу. Поэтому специалисты компании Microsoft рекомендуют вставлять эту инструкцию непосредственно после приведенного выше кода, чтобы установить для счетчика lpddclipper значение, равное 1, как и должно быть.

Следует помнить о том, что прикрепленный к окну отсекатель оказывает влияние только на процесс блиттинга в первичную поверхность, т.е. на содержимое окна. Однако в большинстве случаев для симуляции двойной буферизации создается внеэкранная поверхность, в которую осуществляется блиттинг, после чего эта поверхность с помощью блиттера копируется в первичный буфер (грубая имитация переключения страниц). Таким образом, присоединение отсекателя к первичной поверхности помогает лишь тогда, когда изображение, блиттинг которого осуществляется во внеэкранный буфер, выходит за границы окна. Это происходит только в том случае, когда пользователь изменяет размеры окна.

Работа в 8-битовых оконных режимах

Последняя тема, которой я бы хотел коснуться, посвящена 8-битовому оконному режиму и палитре. Если говорить коротко, нельзя просто создать палитру, выполнить с ней все нужные действия, а затем присоединить ее к первичной поверхности. Необходимо помнить о том, что существует диспетчер палитры (Windows Palette Manager), требующий определенного внимания. Описание работы диспетчера палитры Windows выходит за рамки этой книги, к тому же я не расположен утомлять вас всеми этими скучными подробностями. Суть в том, что при запуске игры в оконном режиме с 256 цветами в нашем распоряжении будет меньшее количество цветов.

Каждое приложение, которое выполняется на рабочем столе, имеет свою логическую палитру, содержащую нужные приложению цвета. Однако *физическая* палитра всего одна, и этот факт имеет большое значение. В физической палитре представлены фактические аппаратные цвета; это соглашение, принятое в Windows. Когда приложение попадает в поле зрения операционной системы, происходит *реализация* логической палитры; другими словами, диспетчер палитры начинает отображать заданные цвета в физическую палитру. Он делает это в силу своих способностей; иногда это у него получается хорошо, а иногда — нет.

Кроме того, от настройки флагов логической палитры будет зависеть, сколько упущений придется сделать системе Windows. Наконец, следует помнить о том, что самой системе необходимо иметь в своем распоряжении хотя бы 20 цветов — 10 первых и 10 последних. Это количество составляет абсолютный минимум, который обеспечивает приемлемый внешний вид приложений Windows. Таким образом, создавая оконное приложение в 256-цветном режиме, приходится ограничиваться 236 цветами (или меньшим их количеством), чтобы осталось место для цветов Windows. Нужно также уметь правильно задавать флаги логической палитры. При такой реализации палитры цвета вашего приложения не будут искажаться операционной системой.

Ниже приведен код, с помощью которого создается настраиваемая палитра. Этот код можно применять для создания в записях от 10-й до 245-й включительно собственных значений цветов в формате RGB. В приведенном коде во все записи заносится серый цвет.

```
LPDIRECTDRAW7 lpdd;                    // Считаем, что эта переменная
                                       // уже настроена
PALETTEENTRY palette[256];             // Хранилище данных палитры
LPDIRECTDRAWPALETTE lpddpal = NULL; // Интерфейс палитры

// Задаем статические записи для Windows;
// дальнейшие действия на них не повлияют
for (int index=0; index<10; index++)
{
  // Первые 10 статических записей:
  palette[index].peFlags = PC_EXPLICIT;
  palette[index].peRed   = index;
  palette[index].peGreen = 0;
```

```
    palette[index].peBlue  = 0;

    // Последние 10 статических записей:
    palette[index+246].peFlags = PC_EXPLICIT;
    palette[index+246].peRed   = index+246;
    palette[index+246].peGreen = 0;
    palette[index+246].peBlue  = 0;
} // for index

// Настройка записей приложения. Их можно загружать
// из файла или другого источника, однако сейчас они
// будут серыми
for (int index=10; index<246; index++)
{
    palette[index].peFlags = PC_NOCOLLAPSE;
    palette[index].peRed   = 64;
    palette[index].peGreen = 64;
    palette[index].peBlue  = 64;
} // for index

// Создание палитры.
if (FAILED(lpdd->CreatePalette(DDPCAPS_8BIT, palette,
    &lpddpal,NULL)))
    { /* ошибка */ }

// Присоединение палитры к первичной поверхности...
```

Обратите внимание на использование флагов PC_EXPLICIT и PC_NOCOLLAPSE. Флаг PC_EXPLICIT означает, что цвета будут отображаться на аппаратные устройства, а флаг PC_NOCOLLAPSE сообщает диспетчеру палитры, чтобы он не пытался отображать данные цвета в другие записи и оставил их в неизменном виде. Если вы хотите выполнить анимацию каких-то цветовых регистров, то можете с помощью побитового оператора OR присоединить флаг PC_RESERVED. При этом менеджер палитры не будет отображать цвета других приложений Windows в ваши записи, так как эти записи могут измениться в любую секунду.

Резюме

Эта глава, без сомнения, была одной из самых длинных в книге. Дело в том, что в ней нужно было изложить очень много материала. Можно было бы продолжать, однако я хочу оставить кое-что на будущее.

В главе описаны все необходимые элементы, включая высокоцветные режимы, блиттер, отсечение, цветовые ключи, теорию дискретизации, оконные приложения DirectDraw, GDI, а также запрос информации в DirectDraw. Кроме того, начато создание библиотеки T3DLIB1.CPP|H, с которой можно ознакомиться прямо сейчас, а можно подождать до конца следующей главы, где описывается каждая функция библиотеки.

В следующей главе я собираюсь сделать небольшой перерыв в изучении DirectDraw (хотя вернусь к нему в конце главы) и изложить некоторые элементы двухмерной геометрии, теории преобразований и растеризации.

ГЛАВА 8

Растеризация векторов и двухмерные преобразования

В этой главе обсуждаются вопросы, касающиеся векторов, способы вычерчивания линий и многоугольников, а также рассматриваются функциональные возможности первого модуля библиотеки T3DLIB, созданием которого завершается данная глава. Здесь вам впервые придется столкнуться с математическими формулами, но это не потребует от вас больших умственных усилий. Самое сложное из описанного здесь — это матрицы. Идя навстречу многочисленным пожеланиям, я намерен также познакомить вас с некоторыми идеями о том, как осуществлять прокрутку и даже реализовать трехмерные изометрические машины. Ниже приведен перечень вопросов, рассматриваемых в данной главе.

- Рисование линий
- Отсечение изображений
- Матрицы
- Двухмерные преобразования
- Рисование многоугольников
- Прокрутка и изометрические трехмерные машины
- Хронометрирование
- Библиотека T3DLIB, версия 1.1

Черчение прямых линий

До настоящего момента вы рисовали графические объекты либо путем вывода одиночных точек, либо при помощи битовых образов. Это невекторные способы представления.

Векторный объект, как показано на рис. 8.1, представляет собой нечто наподобие линии или многоугольника. Давайте вначале рассмотрим, как рисуется линия[1].

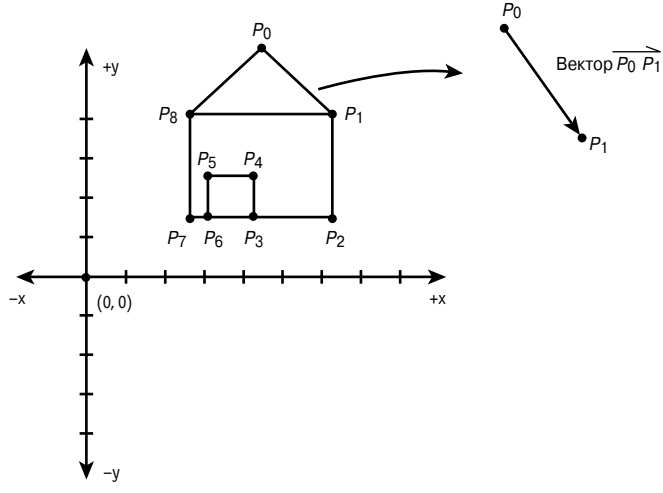


Рис. 8.1. Объект, нарисованный с помощью векторов

На первый взгляд может показаться, что черчение прямой линии — тривиальный процесс, но уверяю вас, что это вовсе не так. Черчение прямой линии на экране компьютера связано с целым рядом проблем, например конечным разрешением, отображением вещественных чисел на сетке, образованной целочисленными значениями, скоростью и т.д. Чаще всего двух- или трехмерное изображение состоит из множества точек, которые определяют многоугольники или видимые поверхности, формирующие объект или сцену. Обычно эти точки задаются с помощью вещественных чисел, например (10.5, 120.3) и т.д.

Основная проблема состоит в том, что экран компьютера представлен в виде двухмерной сетки, образованной целыми числами, и поэтому нарисовать точку (10.5, 120.3) практически невозможно. Ее можно только аппроксимировать. Иногда для этого приходится отбрасывать дробную часть значений координат и изображать точку (10, 120) или же выполнять округление и наносить точку (11, 120). И наконец, можно применить высокотехнологичный метод и воспользоваться функцией отображения взвешенных пикселей, которая выводит несколько пикселей разной яркости, располагая их вокруг центра (10.5, 120.3). Этот способ показан на рис. 8.2.

По сути, фильтр области вычисляет, в какой степени исходный пиксель перекрывается позициями других пикселей, а затем рисует их пиксели тем же цветом, что и интересующий нас пиксель, но с меньшей интенсивностью. В результате получается "закругленный" элемент изображения, контур которого не имеет заметных неровностей, но в целом максимальное разрешение оказывается сниженным.

Вместо долгого и скучного обсуждения алгоритмов вычерчивания прямых линий, фильтрации и подобных вещей, давайте сразу отрежем лучший кусок пирога и рассмотрим парочку действительно хороших алгоритмов рисования прямых, которые вы сможете использовать для вычисления целочисленных координат отрезка, начинающегося в точке (x_0, y_0) и заканчивающегося в точке (x_1, y_1).

[1] Здесь и далее под термином "линия", если явно не оговорено иное, подразумевается прямая линия. — *Прим. ред.*

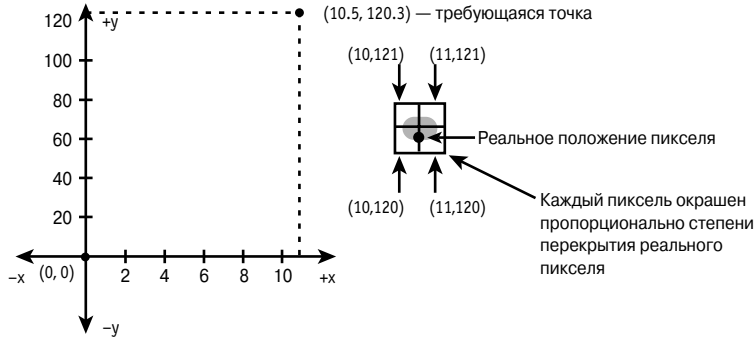


Рис. 8.2. Вывод области одиночного пикселя

Алгоритм Брезенхама

Первый алгоритм, с которым мне бы хотелось вас познакомить, называется *алгоритмом Брезенхама* (по имени человека, который изобрел его в 1965 году). Первоначально этот алгоритм предназначался для вычерчивания линий на графопостроителях, но позже был адаптирован для применения в компьютерной графике. Предлагаю сначала кратко рассмотреть работу этого алгоритма, а затем перейти к описанию его кода. На рис. 8.3 представлена задача, которую мы пытаемся решить: необходимо заполнить те пиксели, начиная от точки p_1 и заканчивая точкой p_2, которые максимально соответствуют реальной линии. Этот процесс называется *растеризацией*.

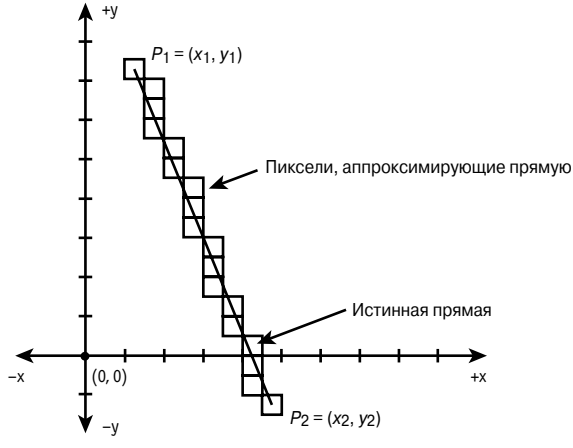


Рис. 8.3. Растеризация линии

Если вы давно не занимались прямыми линиями, позвольте мне освежить вашу память. Наклон прямой определяется углом, который она образует с осью X. Отсюда следует, что прямая с наклоном 0 горизонтальна, тогда как прямая с бесконечным наклоном вертикальна. Прямая с наклоном 1.0 представляет собой линию с наклоном в 45°. Наклон (m) определяется как приращение вдоль оси Y, деленное на приращение вдоль оси X, т.е. математически это записывается так:

$$m = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{y_1 - y_0}{x_1 - x_0}.$$

Например, если прямая проходит через точки $p_0(1,2)$ и $p_1(5,22)$, то ее наклон будет $m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{22-2}{5-1} = \frac{20}{4} = 5$.

Что же на самом деле означает наклон? Он означает, что приращение координаты X на 1.0 изменит координату Y на 5.0. С этого мы и начнем алгоритм растеризации. Теперь вы должны примерно представлять, как рисуется линия.

1. Вычисляется наклон m.
2. Наносится точка (x_0, y_0).
3. Вначале координата X увеличивается на 1.0, а затем увеличивается координата Y — на величину m. Эти значения добавляются к координатам (x_0, y_0).
4. Шаги 2–4 повторяются до тех пор, пока не будет нарисована вся линия.

На рис. 8.4 представлен пример, использующий указанные значения координат точек p_0 и p_1.

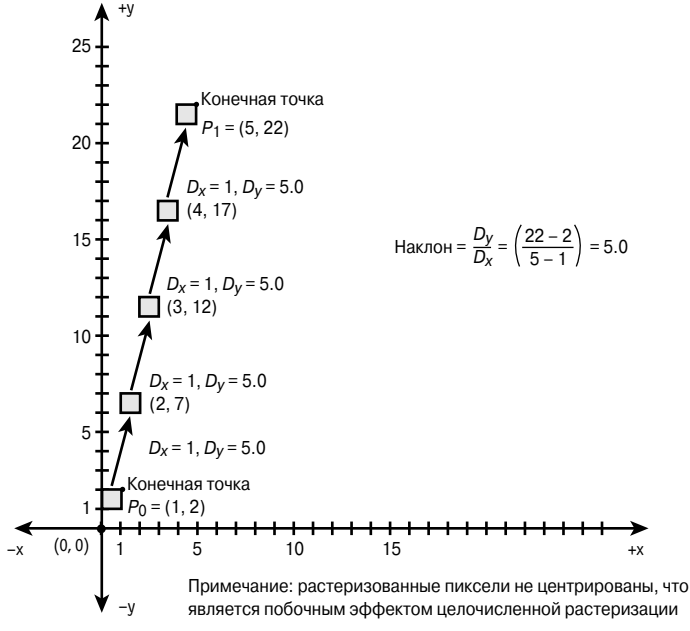


Рис. 8.4. Первая попытка растеризации

Вам понятна идея? Каждый раз, когда вы перемещаетесь вдоль оси x на один пиксель, вы делаете шаг размером в 5.0 пикселей вдоль оси y. Поэтому у вас получится линия, которая содержит множество "дыр". Ошибка состоит в том, что вы не соединяете линию, а наносите пиксели через целочисленные интервалы; т.е. всякий раз значение координаты x является целым числом. В сущности, вы подставляете в уравнение прямой линии $y - y_0 = m(x - x_0)$ целые числа.

Здесь (x, y) представляет собой текущую позицию пикселей, (x_0, y_0) — начальную точку, m — наклон прямой. Выполнив небольшую перестановку, получим $y = m(x - x_0) + y_0$.

Итак, если предположить, что координаты (x_0,y_0) равны $(1,2)$, а m равно 5 (как в предыдущем примере), то получим следующие результаты:

x	y=5*(x-1)+2
1	2 (начальная точка)
2	7
3	12
4	17
5	22 (конечная точка)

Далее возникает вопрос: не может ли оказаться полезной форма уравнения прямой линии, использующая угловой коэффициент и точку пересечения с осьюY: $y = mx + b$.

В этом уравнении b представляет собой точку, в которой линия пересекает ось Y. Однако это решение также не является для вас хоть сколько-нибудь подходящим. Основная проблема заключается в том, что за каждый шаг вы перемещаетесь вдоль оси X на 1.0. Однако это перемещение должно быть намного меньшим (например, равным 0.01) с тем, чтобы вы могли охватить все пиксели линии, не пропуская ни одного. Сообразительный читатель заметит, что в данном случае, сколь бы малым ни было перемещение вдоль оси X, всегда можно найти такое значение наклона, которое приведет к пропускам пикселей. Мы должны попытаться использовать другой способ построения линии, например алгоритм Брезенхама.

Не вдаваясь во все подробности, замечу, что алгоритм Брезенхама начинается в точке (x_0,y_0), но вместо наклона применяется следующий способ: происходит перемещение вдоль оси X на один пиксель, а затем принимается решение о перемещении вдоль оси Y с тем требованием, чтобы проводимая линия в наибольшей степени соответствовала истинной. Для этого используется ошибка, которая следит за точностью растеризуемой линии. Алгоритм непрерывно обновляет ошибку, и его задача состоит в том, чтобы растеризованная линия была как можно ближе к истинной.

Описываемый алгоритм работает в первом квадранте декартовой системы координат; для остальных квадрантов алгоритм получается путем зеркального отражения. Кроме того, алгоритм различает линии двух видов: с наклоном менее 45° (m<1) и с наклоном более 45° (m>1). Я называю их X-доминантными и Y-доминантными соответственно. Итак, напишем псевдокод для X-доминантной линии, которая проходит через точки $p_0(x_0,y_0)$ и $p_1(x_1,y_1)$:

```
// Инициализируем начальную точку
x = x0;
y = y0;

// Вычисляем изменение координат x и y
dx = x1 - x0;
dy = y1 - y0;

// Инициализируем ошибку
error = 0;

// Чертим линию
for (int index = 0; index < dx; index++)
{
  // Наносим один пиксель
  Plot_Pixel(x++, y, color);
```

```
  // Корректируем ошибку
  error += dy;

  // Проверяем ошибку
  if (error > dx)
  {
    // Корректируем ошибку
    error -= dx

    // Переходим к следующей линии
    y--;
  } // if
} // for
```

Это все. Разумеется, данный код справедлив только для первого квадранта, но все остальные могут быть получены путем проверки (и изменения в случае необходимости) знаков и соответствующего изменения значений. Алгоритм остается без изменений.

Перед тем как представить код, мне хотелось бы сделать одно замечание по поводу точности алгоритма. Алгоритм непрерывно минимизирует несоответствие между растеризованной и истинной линией, но, помимо этого, можно немного улучшить и начальные условия. Очевидно, что ошибка начинается со значения 0.0. На самом деле это не совсем правильно. Было бы гораздо лучше, если каким-либо образом можно было бы учитывать позицию первого пикселя и задавать ошибку с большей точностью. С этой целью ошибке присваивается значение 0.5, но поскольку мы используем целые числа, эта величина должна умножаться на 2 и затем добавляться к приращениям d_x и d_y. Тогда в окончательном алгоритме ошибка будет задаваться примерно так:

```
// x-доминантная линия
error = 2*dy - dx

// y-доминантная линия
error = 2*dx - dy
```

Напишем окончательный алгоритм, реализованный в виде функции под названием Draw_Line(), которая находится в нашей библиотеке. Эта функция принимает в качестве параметров обе конечные точки, цвет, указатель на видеобуфер, шаг видеопамяти и рисует прямую линию.

```
int Draw_Line(int x0, int y0,     // Начальная точка
        int x1, int y1,           // Конечная точка
        UCHAR  color,             // Индекс цвета
        UCHAR  *vb_start,         // Видеобуфер
        int    lpitch )           // Шаг видеопамяти
{
// Эта функция рисует линию по алгоритму Брезенхама.
// Линия начинается в точке x0,y0 и
// заканчивается в точке x1,y1

int dx,      // Разность значений по оси x
  dy,        // Разность значений по оси y
  dx2,       // dx,dy * 2
  dy2,
  x_inc,     // Шаг перемещения во время рисования
```

```
    y_inc,     // Шаг перемещения во время рисования
    error,     // Ошибка (переменная принятия решения)
    index;     // Переменная цикла

// Вычисляем адрес первого пикселя в видеобуфере
vb_start = vb_start + x0 + y0*lpitch;

// Рассчитываем смещение по горизонтали и вертикали
dx = x1 - x0;
dy = y1 - y0;

// Проверяем направление линии (угол наклона)
if (dx >= 0)
{
   x_inc = 1;
} // линия направлена вправо
else
{
   x_inc = -1;
   dx = -dx; // Нужна абсолютная величина
} // Линия направлена влево

// проверяем y-составляющую наклона
if (dy >= 0)
{
   y_inc = lpitch;
}  // Линия направлена вниз
else
{
   y_inc = -lpitch;
   dy = -dy; // Нужна абсолютная величина
} // Линия направлена вверх

// Вычисляем (dx,dy) * 2
dx2 = dx << 1;
dy2 = dy << 1;

// Теперь рисуем линию с учетом того, вдоль какой
// оси приращение больше
if (dx > dy)
{
   // инициализируем ошибку
   error = dy2 - dx;

   // Чертим линию
   for (index = 0; index <= dx; index++)
   {
      // Выводим пиксель
      *vb_start = color;

      // Проверяем, нет ли переполнения ошибки
      if (error >= 0)
      {
```

```
        error -= dx2;
        // переходим к следующей линии
        vb_start += y_inc;
      } // if

      // Корректируем ошибку
      error += dy2;

      // Переходим к следующему пикселю
      vb_start += x_inc;

    } // for
}
else
{
    // Инициализируем ошибку
    error = dx2 - dy;

    // Рисуем линию
    for (index = 0; index <= dy; index ++)
    {
      // Выводим пиксель
      *vb_start = color;

      // Проверяем, нет ли переполнения ошибки
      if (error >= 0)
      {
        error -= dy2;
        // Переходим к следующей линии
        vb_start += x_inc;
      }

      // Корректируем ошибку
      error += dx2;

      // переходим к следующему пикселю
      vb_start += y_inc;

    } // for
} // else

// Возвращаем код успешного завершения функции
return (1);

} // конец функции Draw_Line
```

НА ЗАМЕТКУ Эта функция работает только в 8-битовых режимах. В библиотеке имеется и 16-битовая версия, которая называется Draw_Line16().

Данную функцию условно можно разделить на три основные части. В первой определяются все знаки, задаются конечные точки и вычисляются приращения вдоль осей X и Y. Далее проверяется доминантность линии (т.е. какое из условий — dx>dy или dx<=dy — истинно), а затем вычерчивается сама линия.

Повышение скорости алгоритма

При беглом взгляде на код может показаться, что он достаточно компактен и не подлежит дальнейшей оптимизации. Тем не менее существует ряд способов повышения скорости этого алгоритма. Один из них — это учет симметричности линий относительно их центральных точек (что показано на рис. 8.5), это избавляет от необходимости выполнять алгоритм для всей линии. Нужно лишь разделить линию пополам, нарисовать одну ее половину, а затем просто зеркально отобразить другую половину. Теоретически подобный подход выглядит убедительно, но реализация связана с определенными трудностями, поскольку приходится отдельно рассматривать линии с четным и нечетным количеством пикселей и решать вопрос вывода при необходимости дополнительного пикселя. Это не только сложно, но и очень некрасиво.

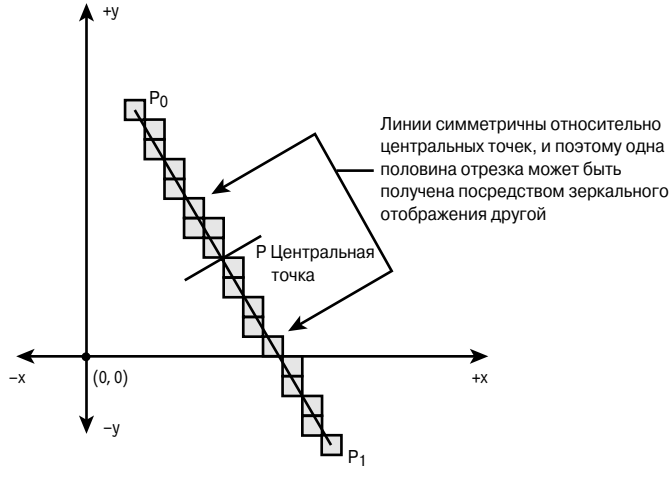


Рис. 8.5. Линии симметричны относительно своих центральных точек

Некоторыми авторами были предложены и другие варианты оптимизации. К ним можно отнести алгоритм Майкла Абраша (Michael Abrash) Run-Slicing, алгоритм Ксиалона By (Xialon Wu) Symmetric Double Step и алгоритм Рокне (Rokne) Quadruple Step. В основном все эти алгоритмы используют свойства тех или иных конфигураций пикселей, образующих линию. Например, алгоритм Run-Slicing основан на том, что иногда существуют очень длинные последовательности пикселей с одной координатой, как, например, в линии между точками (0,0) и (100,1), которая должна содержать две горизонтальные линии по 50 пикселей, что наглядно показано на рис. 8.6.

Основная проблема применения данного алгоритма — сложная настройка; при этом реальную выгоду он приносит только при выводе линий с очень большими или очень малыми углами наклона.

Алгоритм Symmetric Double Step работает на основе аналогичных предпосылок, но вместо рассмотрения линий он принимает во внимание тот факт, что для любой пары пикселей (рис. 8.7) возможны только четыре различных шаблона их размещения. Вычислив ошибку, можно легко определить следующий шаблон и вывести его целиком, двигаясь, по сути, с удвоенной (по отношению к обычной) скоростью.

Демонстрационная программа рисования линий, находящаяся на прилагаемом компакт-диске, называется `DEMO8_1.CPP` и выводит произвольные линии в режиме экрана 640×480.

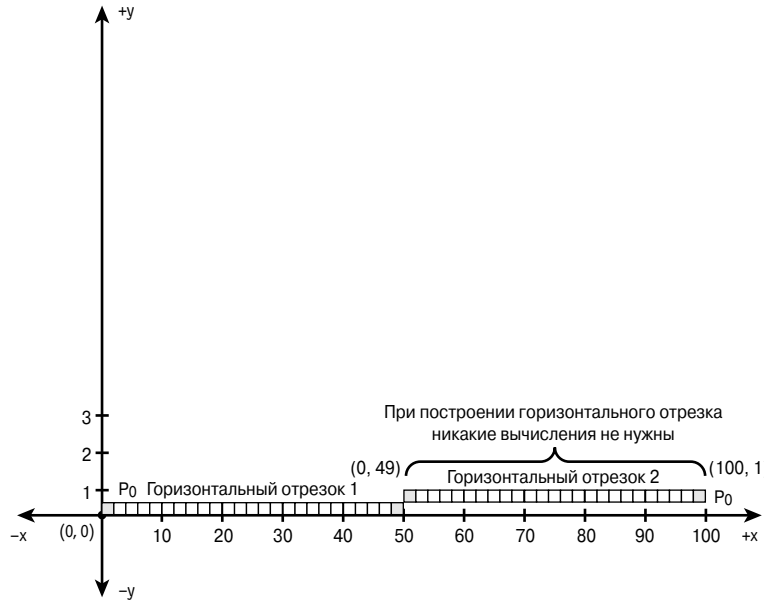


Рис. 8.6. Вывод линии по алгоритму Run-Slicing

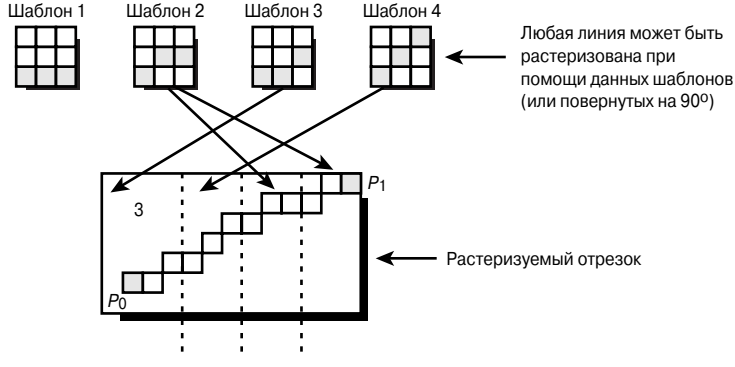


Рис. 8.7. Вычерчивание линий с помощью растра

НА ЗАМЕТКУ

Если я решу, что нам необходим более быстрый способ вычерчивания линий, я опишу его. Но сейчас мне не хочется излишне усложнять материал, так что мы переходим к отсечениям изображений.

Основы отсечения двухмерных изображений

Компьютерные изображения могут отсекаться двумя способами: на уровне пространства изображений и на уровне пространства объектов. *На уровне пространства изображений* на самом деле означает "на уровне пикселей". При растеризации изображения имеется отсекающий фильтр, который и определяет, попадает ли некоторый пиксель в смот-

ровое окно. Этот метод подходит для вывода одиночных пикселей, но не для больших объектов, например растровых изображений, линий или многоугольников. Для рисования подобных объектов, обладающих некоторой геометрической формой, можно воспользоваться дополнительной информацией.

Предположим, вам нужно написать отсекатель растрового изображения. Для этого требуется отсечь один прямоугольник другим, т.е. оставить от выводимого прямоугольника только ту область, которая будет видна в ограничивающем прямоугольнике. В данном случае все очень просто: область, которую вы должны передать для блиттирования, представляет собой пересечение этих прямоугольников.

Как ни странно, но отсекать линии сложнее. На рис. 8.8 показана общая задача отсечения линии для прямоугольного смотрового окна.

Как видите, существует четыре общих случая.

- Случай 1. Вся линия находится за пределами области отсечения — тривиальный (и очень легкий) случай отсечения.
- Случай 2. Линия полностью попадает в область отсечения. В этом случае отсечение не требуется и линия передается для растеризации целиком.
- Случай 3. Один конец линии выходит за рамки области отсечения и должен быть отсечен.
- Случай 4. Оба конца линии выходят за рамки области отсечения и должны быть отсечены.

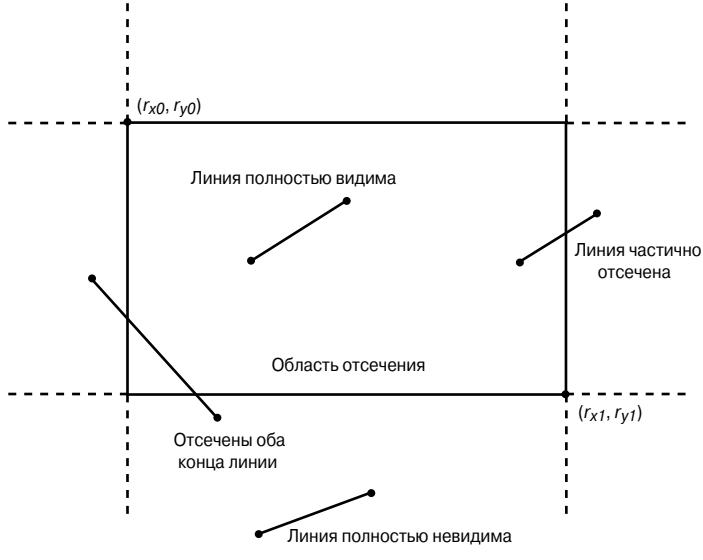


Рис. 8.8. Общая проблема, связанная с отсечением линии

Для отсечения линий используется несколько известных алгоритмов, например алгоритмы Кохена–Сазерленда (Cohen–Sutherland), Сайруса–Бека (Cyrus–Beck) и др. Но, прежде чем рассматривать какой-либо из них, попытайтесь проделать подобные вычисления самостоятельно.

Пусть в двухмерном пространстве у вас есть линия, начинающаяся в точке (x_0, y_0) и заканчивающаяся в точке (x_1, y_1), а также прямоугольное смотровое окно, заданное вершинами (r_{x0}, r_{y0}) и (r_{x1}, r_{y1}), и вам нужно отсечь линию, ограничив ее рамками ука-

занной области. Итак, вам потребуется препроцессор (или отсекающий фильтр, если хотите), принимающий входные значения (x_0, y_0), (x_1, y_1), (r_{x0}, r_{y0}) и (r_{x1}, r_{y1}) и выдающий в результате новый отрезок линии, который начинается в точке (x_0', y_0'), заканчивается в точке (x_1', y_1') и определяет видимую часть линии. Этот процесс показан на рис. 8.9. Когда сталкиваешься с подобной проблемой, понятно, что без вычисления точки пересечения двух линий не обойтись. Итак, начнем именно с этой фундаментальной задачи.

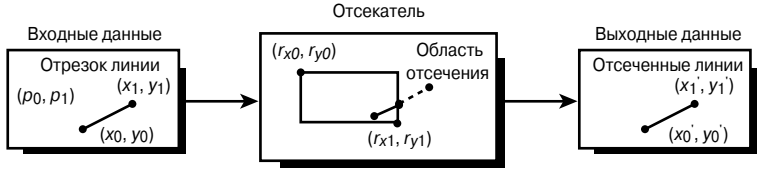


Рис. 8.9. Схематическое представление процесса отсечения

Отрезок линии, начинающийся в точке (x_0, y_0) и заканчивающийся в точке (x_1, y_1), должен пересечь одну из сторон отсекающего прямоугольника: левую, правую, верхнюю или нижнюю. Это, означает, что вам не нужно искать точку пересечения двух произвольно направленных линий, поскольку растеризуемая линия всегда будет пересекать либо горизонтальную, либо вертикальную линию. Плохо это или хорошо, но данная информация известна вам изначально. Для вычисления точки пересечения двух линий используется несколько методов, и все они основаны на математическом уравнении линий.

Линии могут описываться с помощью приведенных ниже уравнений.

На основе точки пересечения оси Y	$y = mx + b$
На основе наклона линии	$(y - y_0) = m(x - x_0)$
На основе двух точек	$(y - y_0) = (x - x_0)(y_1 - y_0)/(x_1 - x_0)$
Обобщенное уравнение	$ax + by = c$
Параметрическая форма	$\mathbf{P} = \mathbf{p}_0 + \mathbf{V}t$

СОВЕТ

Не стоит расстраиваться, если использование уравнения в параметрической форме кажется вам несколько сложным; немного позже я поясню, что собой представляют параметрические уравнения. Кроме того, замечу, что уравнение прямой, использующее одну точку и наклон, и уравнение, использующее две точки, представляют собой одно и то же уравнение (так как $m = (y_1-y_0)/(x_1-x_0)$).

Вычисление точки пересечения с использованием одной точки и наклона

Мне нравятся два вида уравнения прямой — с использованием одной точки и наклона и обобщенное уравнение, так что я буду рассматривать оба варианта записи. Давайте вычислим точку пересечения линий p_0 и p_1, используя уравнения этих двух типов. Это немного подготовит вас к восприятию действительно сложных математических формул, с которыми вы столкнетесь в последующих главах. Вначале давайте обратимся к уравнению, использующему одну точку и наклон прямой (рис. 8.10).

Итак, пусть первый отрезок p_0 начинается в точке (x_0, y_0) и заканчивается в точке (x_1, y_1), а второй отрезок p_1 начинается в точке (x_2, y_2) и заканчивается в точке (x_3, y_3). Отрезки p_0 и p_1 могут иметь произвольное направление.

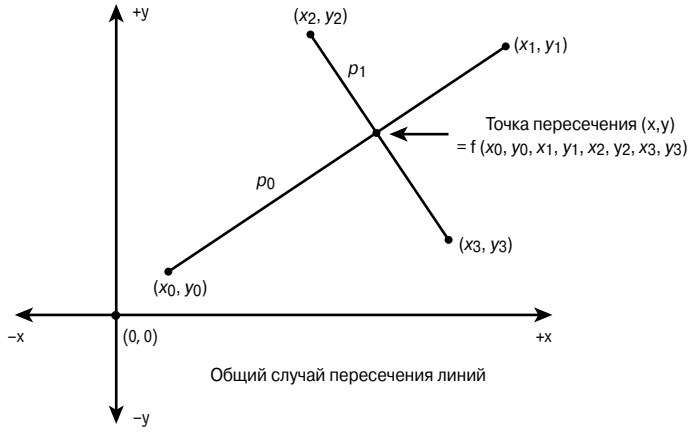


Рис. 8.10. Вычисление точки пересечения двух линий

Наклон и уравнение первой прямой записываются как

$$m_0 = \frac{(y_1 - y_0)}{(x_1 - x_0)}, \; y - y_0 = m_0 (x - x_0). \tag{1}$$

Для второй линии уравнения выглядят как

$$m_1 = \frac{(y_3 - y_2)}{(x_3 - x_2)}, \; y - y_2 = m_1 (x - x_2). \tag{2}$$

Итак, мы получаем два уравнения с двумя неизвестными:

$$y - y_0 = m_0 (x - x_0);$$
$$y - y_2 = m_1 (x - x_2).$$

Найти точку пересечения (x,y) можно двумя основными способами: с помощью подстановки и матричных операций. Вначале попытаемся найти решение методом подстановки. Идея заключается в том, чтобы выразить одну переменную через другую и в таком виде подставить ее в другое уравнение. Попробуем выразить y через x из первого уравнения:

$$y = y_0 + m_0 (x - x_0).$$

Подставляя его в уравнение (2), получаем

$$y_0 + m_0 (x - x_0) - y_2 = m_1 (x - x_2),$$

откуда находим

$$x = \frac{m_0 x_0 - m_1 x_2 - y_0 + y_2}{m_0 - m_1}. \tag{3}$$

Подставляя найденное значение x в первое уравнение, находим y:

$$\begin{aligned} y &= y_0 + m_0 \left(\frac{m_0 x_0 - m_1 x_2 - y_0 + y_2}{m_0 - m_1} - x_0 \right) \\ &= y_0 + m_0 \frac{y_2 - y_0 + m_1 (x_0 - x_2)}{m_0 - m_1}. \end{aligned} \tag{4}$$

Теперь необходимо рассмотреть ряд вопросов, которые не должны ускользнуть от вашего внимания. Первое, существуют ли такие ситуации, когда применение описанных уравнений связано с проблемами? Конечно. В высшей математике использование бесконечности не представляет проблемы, чего нельзя сказать о компьютерной графике. В уравнениях (3) и (4) выражения $(m_0 - m_1)$ могут оказаться равны 0.0, если линии имеют равный наклон, иными словами, если они параллельны.

В этом случае линии никогда не пересекутся, знаменатель будет равен 0.0, и результат деления в уравнениях (3) и (4) стремится к бесконечности. Формально это означает, что линии пересекутся в бесконечности, но так как вы работаете исключительно с экраном, имеющим разрешение не выше 1024×768 или около того, рассмотрение такого случая не входит в наши планы.

Главный вывод: уравнения пересечения линий справедливы только для тех линий, которые действительно пересекаются. Если они никогда не пересекутся, наши формулы не срабатывают. Условие пересечения легко проверяется. Перед началом вычислений просто проверяйте m_0 и m_1 на равенство. Если m_0 равно m_1, точки пересечения не существует.

Канонический вид уравнения прямой

Общая форма линейного уравнения имеет вид $ax + by = c$ или, если вы предпочитаете каноническую форму, $ax + by + c = 0$.

В действительности все уравнения прямой могут быть приведены к общему виду. Например, если в уравнении прямой с точкой пересечения оси Y сделать перестановку: $y = mx + b$, его можно записать в каноническом виде как $mx + (-1)y - b = 0$, т.е. можно считать, что $a = m$, b = -1 и c = -b. Если же у нас даны две точки прямой (x_0, y_0) и (x_1, y_1), то и в этом случае уравнение прямой

$$(y - y_0) = (x - x_0)(y_1 - y_0)/(x_1 - x_0)$$

можно преобразовать к каноническому виду:

$$x(y_1 - y_0) + y(x_0 - x_1) + y_0x_1 - y_1x_0 = 0.$$

Вычисление пересечения двух линий с помощью матриц

Теперь, когда вы знаете, как преобразовать любое уравнение прямой к каноническому виду, можно приступить к рассмотрению метода поиска точки пересечения линий, основанного на использовании матриц.

Пусть заданы два линейных уравнения следующего вида:

$$a_1x + b_1y = c_1;$$
$$a_2x + b_2y = c_2.$$

Нам нужно найти такие значения (x,y), которые являются решениями обоих уравнений. В предыдущем примере применялась подстановка, можно применить ее и сейчас, но я хочу показать вам метод решения системы линейных уравнений с помощью матриц. Я не намерен глубоко погружаться в теорию, поэтому сразу приведу конечный результат и расскажу, как найти (x,y) с помощью матричных операций. Итак, пусть матрица A равна

$$A = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix},$$

вектор неизвестных переменных X имеет вид

$$X = \begin{bmatrix} x \\ y \end{bmatrix},$$

а вектор свободных членов (констант)

$$Y = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Таким образом, систему наших уравнений можно записать как матричное уравнение $AX = Y$. Умножив обе части на обратную к A матрицу, получим

$$A^{-1}AX = X = A^{-1}Y.$$

Вот и все. Разумеется, вы должны знать, как находить обратную матрицу, а затем для получения (x, y) выполнить умножение матриц, но я вам немного помогу и сразу покажу окончательные результаты:

$$x = \frac{\det A_1}{\det A},$$
$$y = \frac{\det A_2}{\det A},$$

где A_1 и A_2 представляют собой следующие матрицы:

$$A_1 = \begin{bmatrix} c_1 & b_1 \\ c_2 & b_2 \end{bmatrix},\ A_2 = \begin{bmatrix} a_1 & c_1 \\ a_2 & c_2 \end{bmatrix}.$$

По сути, мы заменили первый и второй столбцы матрицы A вектором Y и создали соответственно матрицы A_1 и A_2. *Определитель* (det M) матрицы M для произвольной матрицы M размером 2×2

$$M = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

вычисляется следующим образом:

$$\det M = (ad - bc).$$

Чтобы лучше понять этот материал, рассмотрим конкретный пример.

$$\begin{cases} 5x - 2y = -1, \\ 2x + 3y = 3. \end{cases}$$

$$A = \begin{bmatrix} 5 & -2 \\ 2 & 3 \end{bmatrix},\ X = \begin{bmatrix} x \\ y \end{bmatrix},\ Y = \begin{bmatrix} -1 \\ 3 \end{bmatrix}.$$

Соответственно

$$A_1 = \begin{bmatrix} -1 & -2 \\ 3 & 3 \end{bmatrix} \text{ и } A_2 = \begin{bmatrix} 5 & -1 \\ 2 & 3 \end{bmatrix}$$

и окончательное решение системы уравнений будет таким:

$$x = \frac{\det\begin{bmatrix} -1 & -2 \\ 3 & 3 \end{bmatrix}}{\det\begin{bmatrix} 5 & -2 \\ 2 & 3 \end{bmatrix}} = \frac{-3+6}{15+4} = \frac{3}{19};$$

$$y = \frac{\det\begin{bmatrix} 5 & -1 \\ 2 & 3 \end{bmatrix}}{\det\begin{bmatrix} 5 & -2 \\ 2 & 3 \end{bmatrix}} = \frac{15+2}{15+4} = \frac{17}{19}.$$

Выглядит ужасно, не правда ли? Да, но все это программирование игр — сплошная математика, особенно в наши дни. К счастью, если вы один раз написали код, выполняющий все необходимые математические вычисления, вам больше не нужно беспокоиться по этому поводу. Но неплохо было бы понимать его, именно поэтому я и позволил себе краткое напоминание азов.

Совершив небольшой экскурс в область математики, вернемся к тому, с чего начинали, — к отсечению.

Отсечение линии

Вполне очевидно, что понятие отсечения является тривиальным, но практическая реализация может оказаться несколько сложнее, так как здесь в игру вступает линейная алгебра. По меньшей мере, вы должны понимать, как работать с уравнениями прямых линий и вычислять их пересечение. Однако, как я уже говорил, вы всегда можете воспользоваться наличием априорной информации, позволяющей упростить математические методы. Отсечение — как раз одна из таких ситуаций.

До окончательного отсечения линии некоторым прямоугольником нам еще предстоит пройти долгий путь, но мы добьемся своего. А сейчас обратимся к задаче и посмотрим, нельзя ли упростить вычисления, исходя из того, что одна из пересекающихся линий вертикальна (либо горизонтальна). Взгляните на рис. 8.11.

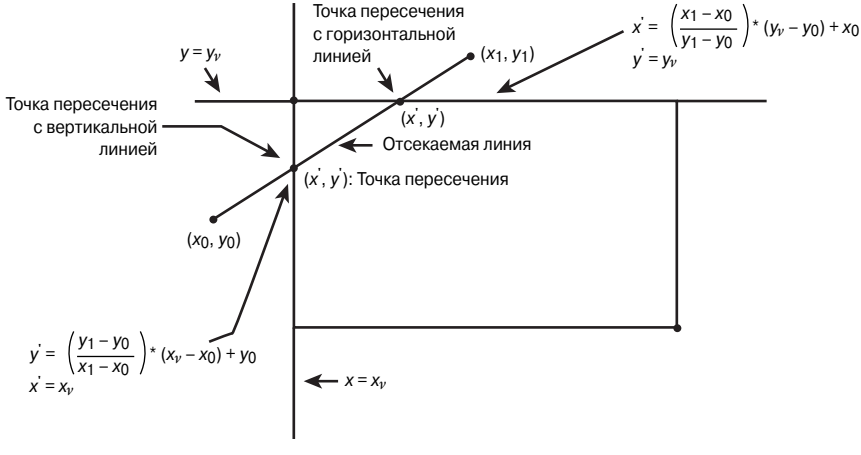


Рис. 8.11. Отсечение линии прямоугольником гораздо проще общего случая

Как следует из рисунка, достаточно рассмотреть только одну переменную, что значительно упрощает математические вычисления. Вместо трудоемких математических операций поиска точки пересечения проще всего обратиться к уравнению прямой, использующей одну точку и наклон, и подставить в него известное значение любой из линий вида $X = \text{const}$ или $Y = \text{const}$. Пусть, например, отсекающая область задается точками

(x_1,y_1) и (x_2,y_2). Если вам нужно узнать, в какой точке линия пересекает левую границу, задача облегчается тем, что в точке пересечения x-координата должна быть равна x_1. Таким образом, вам нужно найти координату y по известной координате x, и дело сделано.

Если же вы хотите найти точку пересечения с горизонтальной линией, например с нижней границей отсекающего прямоугольника (y_2 в нашем случае), то вам известно, что значение координаты y равно y_2, поэтому остается только найти x. Опишем математические операции, выполняемые при вычислении точек пересечения (x,y) прямой, проходящей через точки (x_0,y_0) и (x_1,y_1), с горизонтальной линией $y = y_h$ и вертикальной линией $x = x_v$.

Точка пересечения с горизонтальной линией имеет вид (x,y_h), и нам требуется найти x.

Уравнение прямой, проходящей через две точки, имеет вид

$$(y - y_0) = (x - x_0)(y_1 - y_0)/(x_1 - x_0).$$

Подставляя в это уравнение значение $y = y_h$, находим

$$x = x_0 + (y_h - y_0)(x_1 - x_0)/(y_1 - y_0).$$

Теперь вы умеете вычислять точки пересечения прямой линии с другой произвольной прямой, а также вертикальной или горизонтальной линией (что важно в случае прямоугольного отсечения), поэтому можно переходить к другим вопросам проблемы отсечения.

Алгоритм Кохена–Сазерленда

Вообще говоря, вы должны выяснить, будет ли линия видима полностью, частично видима, частично отсечена (с одного конца) или полностью отсечена (с обоих концов). Этот вопрос вполне решаем, и существует ряд алгоритмов, позволяющих обрабатывать все эти ситуации. Самое широкое распространение получил алгоритм Кохена–Сазерленда (Cohen–Sutherland). Он работает достаточно быстро, неплохо реализуется, и ему посвящены многочисленные публикации.

В сущности, этот алгоритм достаточно прост. Но вместо использования миллиона условных операторов он разбивает отсекающую область на несколько секторов, а затем присваивает битовый код каждой конечной точке отсекаемого отрезка. Затем ситуация идентифицируется с помощью всего лишь нескольких команд if или case. На рис. 8.12 графически представлен план решения этой задачи.

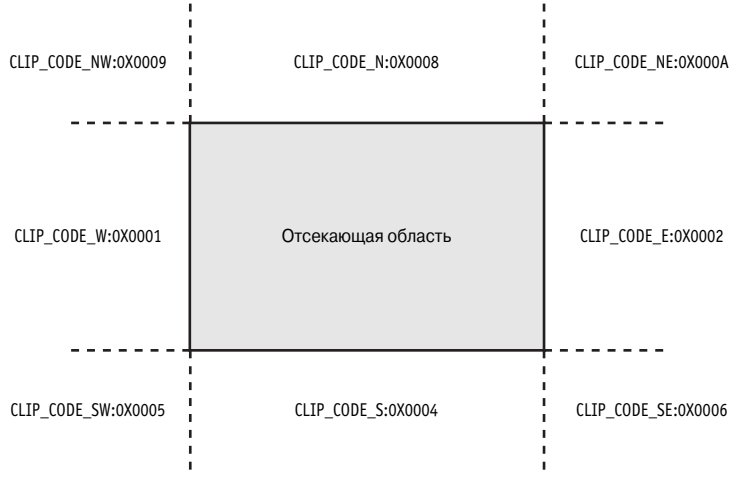


Рис. 8.12. Использование кодов отсечения для эффективного определения конечных точек линии

Следующая функция представляет собой написанную мною версию алгоритма Кохена–Сазерленда.

```
int Clip_Line(int &x1, int &y1, int &x2, int &y2)
{
  // Данная функция отсекает переданную ей линию
  // глобально определенной отсекающей областью

  // Внутренние коды отсечения
#define CLIP_CODE_C 0x0000
#define CLIP_CODE_N 0x0008
#define CLIP_CODE_S 0x0004
#define CLIP_CODE_E 0x0002
#define CLIP_CODE_W 0x0001

#define CLIP_CODE_NE 0x000a
#define CLIP_CODE_SE 0x0006
#define CLIP_CODE_NW 0x0009
#define CLIP_CODE_SW 0x0005

  int xc1 = x1,
  yc1 = y1,
  xc2 = x2,
  yc2 = y2;

  int p1_code = 0,
    p2_code = 0;

  // Определим коды для p1 и p2
  if (y1 < min_clip_y)
    p1_code |= CLIP_CODE_N;
  else
    if (y1 > max_clip_y)
      p1_code |= CLIP_CODE_S;

  if (x1 < min_clip_x)
    p1_code |= CLIP_CODE_W;
  else
    if (x1 > max_clip_x)
      p1_code |= CLIP_CODE_E;

  if (y2 < min_clip_y)
    p1_code |= CLIP_CODE_N;
  else
    if (y2 > max_clip_y)
      p1_code |= CLIP_CODE_S;

  if (x2 < min_clip_x)
    p2_code |= CLIP_CODE_W;
  else
    if (x2 > max_clip_x)
      p2_code |= CLIP_CODE_E;
```

```
// Проверяем тривиальный случай отбрасывания
// всей линии
if (( p1_code & p2_code ))
   return (0);

// Проверяем, видна ли линия полностью; если да,
// оставляем точки нетронутыми
if ( p1_code == 0  &&  p2_code == 0 )
   return (1);

// determine end clip point for p1
switch(p1_code)
{
   case CLIP_CODE_C: break;

   case CLIP_CODE_N:
      {
         yc1=min_clip_y;
         xc1=x1+0.5+(min_clip_y-y1)*(x2-x1)/(y2-y1);
      } break;
   case CLIP_CODE_S:
      {
         yc1=max_clip_y;
         xc1=x1+0.5+(max_clip_y-y1)*(x2-x1)/(y2-y1);
      } break;

   case CLIP_CODE_W:
      {
         xc1=min_clip_x;
         yc1=y1+0.5+(min_clip_x-x1)*(y2-y1)/(x2-x1);
      } break;

   case CLIP_CODE_E:
      {
         xc1=max_clip_x;
         yc1=y1+0.5+(max_clip_x-x1)*(y2-y1)/(x2-x1);
      } break;

      // Более сложные случаи - с двумя пересечениями
   case CLIP_CODE_NE:
      {
         // Северная горизонталь
         yc1=min_clip_y;
         xc1=x1+0.5+(min_clip_y-y1)*(x2-x1)/(y2-y1);

         if (xc1 < min_clip_x || xc1 > max_clip_x)
         {
            // Восточная вертикаль
            xc1=max_clip_x;
            yc1=y1+0.5+
               (max_clip_x-x1)*(y2-y1)/(x2-x1);
         } // if

      } break;
```

```
    case CLIP_CODE_SE:
      {
        // Южная горизонталь
        yc1=max_clip_y;
        xc1=x1+0.5+(max_clip_y-y1)*(x2-x1)/(y2-y1);

        if (xc1 < min_clip_x || xc1 > max_clip_x)
        {
          // Восточная вертикаль
          xc1=max_clip_x;
          yc1=y1+0.5+
            (max_clip_x-x1)*(y2-y1)/(x2-x1);
        } // if

      } break;

    case CLIP_CODE_NW:
      {
        // Северная горизонталь
        yc1=min_clip_y;
        xc1=x1+0.5+(min_clip_y-y1)*(x2-x1)/(y2-y1);

        if (xc1 < min_clip_x || xc1 > max_clip_x)
        {
          xc1=min_clip_x;
          yc1=y1+0.5+
            (min_clip_x-x1)*(y2-y1)/(x2-x1);
        } // if

      } break;

    case CLIP_CODE_SW:
      {
        // Южная горизонталь
        yc1=max_clip_y;
        xc1=x1+0.5+(max_clip_y-y1)*(x2-x1)/(y2-y1);

        if (xc1 < min_clip_x || xc1 > max_clip_x)
        {
          xc1=min_clip_x;
          yc1=y1+0.5+
            (min_clip_x-x1)*(y2-y1)/(x2-x1);
        } // if

      } break;

    default: break;

  } // switch

  // Точка отсечения со стороны p2
  switch(p2_code)
```

```
{
  case CLIP_CODE_C: break;

  case CLIP_CODE_N:
    {
      yc2 = min_clip_y;
      xc2 = x2+(min_clip_y-y2)*(x1-x2)/(y1-y2);
    } break;

  case CLIP_CODE_S:
    {
      yc2 = max_clip_y;
      xc2 = x2+(max_clip_y-y2)*(x1-x2)/(y1-y2);
    } break;

  case CLIP_CODE_W:
    {
      xc2 = min_clip_x;
      yc2 = y2+(min_clip_x-x2)*(y1-y2)/(x1-x2);
    } break;

  case CLIP_CODE_E:
    {
      xc2 = max_clip_x;
      yc2 = y2+(max_clip_x-x2)*(y1-y2)/(x1-x2);
    } break;

  case CLIP_CODE_NE:
    {
      yc2=min_clip_y;
      xc2=x2+0.5+(min_clip_y-y2)*(x1-x2)/(y1-y2);

      if (xc2 < min_clip_x || xc2 > max_clip_x)
      {
        xc2=max_clip_x;
        yc2=y2+0.5+
          (max_clip_x-x2)*(y1-y2)/(x1-x2);
      } // if

    } break;

  case CLIP_CODE_SE:
    {
      yc2=max_clip_y;
      xc2=x2+0.5+(max_clip_y-y2)*(x1-x2)/(y1-y2);

      if (xc2 < min_clip_x || xc2 > max_clip_x)
      {
        xc2=max_clip_x;
        yc2=y2+0.5+
          (max_clip_x-x2)*(y1-y2)/(x1-x2);
      } // if
```

```
        } break;

      case CLIP_CODE_NW:
        {
          yc2=min_clip_y;
          xc2=x2+0.5+(min_clip_y-y2)*(x1-x2)/(y1-y2);

          if (xc2 < min_clip_x || xc2 > max_clip_x)
          {
            xc2=min_clip_x;
            yc2=y2+0.5+
              (min_clip_x-x2)*(y1-y2)/(x1-x2);
          } // if

        } break;

      case CLIP_CODE_SW:
        {
          yc2=max_clip_y;
          xc2=x2+0.5+(max_clip_y-y2)*(x1-x2)/(y1-y2);

          if (xc2 < min_clip_x || xc2 > max_clip_x)
          {
            xc2=min_clip_x;
            yc2=y2+0.5+
              (min_clip_x-x2)*(y1-y2)/(x1-x2);
          } // if

        } break;

      default:break;

    } // switch

    // Проверка границ
    if ((xc1 < min_clip_x) || (xc1 > max_clip_x) ||
       (yc1 < min_clip_y) || (yc1 > max_clip_y) ||
       (xc2 < min_clip_x) || (xc2 > max_clip_x) ||
       (yc2 < min_clip_y) || (yc2 > max_clip_y) )
    {
       return(0);
    } // if

    // Сохраняем переменные
    x1 = xc1;
    y1 = yc1;
    x2 = xc2;
    y2 = yc2;

    return(1);

} // Clip_Line
```

Вы должны передать функции конечные точки отрезка, и функция отсечет его в соответствии с отсекающим прямоугольником, информация о котором хранится в глобальных переменных

```
int  min_clip_x = 0,   // Отсекающий прямоугольник
   max_clip_x = SCREEN_WIDTH-1,
   min_clip_y = 0,
   max_clip_y = SCREEN_HEIGHT-1;
```

Обычно я присваиваю этим переменным значения, соответствующие размерам экрана. В данной функции следует отметить одну деталь: в качестве параметров она принимает ссылки, а следовательно, значения переменных могут быть изменены. Следующий пример поясняет использование этой функции:

```
// Отсечем отрезок, начинающийся в точке (x1,y1) и
// заканчивающийся в точке (x2,y2)

// Сделаем копии конечных точек отрезка
int  clipped_x1 = x1,
   clipped_y1 = y1,
   clipped_x2 = x2,
   clipped_y2 = y2;

// Отсекаем отрезок
Clip_Line (clipped_x1, clipped_y1,
      clipped_x2, clipped_y2);
```

Когда функция возвратит переменные clipped_*, в них будут содержаться новые значения, полученные в результате отсечения линии прямоугольником (информация о котором содержится в глобальных переменных).

В ходе выполнения демонстрационной программы DEMO8_2.CPP создается отсекающая область размером 200×200, расположенная в центре экрана, а затем внутри нее вычерчиваются произвольные линии. Вам остается понаблюдать, как они отсекаются.

Каркасные многоугольники

Теперь, когда вы знаете, как рисуются и отсекаются линии, вы готовы к рассмотрению объектов более высокого порядка, например многоугольников. На рис. 8.13 показаны некоторые примеры многоугольников: треугольник, квадрат и пятиугольник. Многоугольник состоит из трех и более соединенных отрезками точек и представляет собой замкнутую ломаную линию. Многоугольники могут быть выпуклыми или невыпуклыми. Существует математическое определение и соответствующее доказательство, позволяющие выяснить, выпуклый или невыпуклый многоугольник перед вами. А если говорить проще, то выпуклый многоугольник не имеет “вмятин”, чего нельзя сказать о невыпуклом.

А теперь познакомимся с некоторыми идеями о том, как представлять двухмерные многоугольные объекты и манипулировать ими.

Один из способов проверки невыпуклости многоугольника заключается в следующем: если есть две вершины многоугольника, такие, что соединяющий их отрезок будет выходит за пределы многоугольника, то такой многоугольник является невыпуклым.

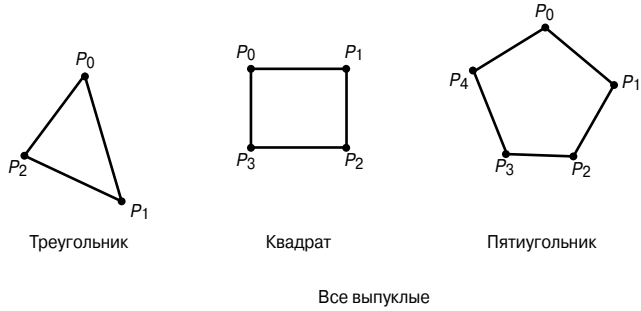


Рис. 8.13. Некоторые основные многоугольники

Структуры данных многоугольников

При программировании игр выбор структуры данных имеет огромное значение. Забудьте обо всем, что вы когда-либо знали об использовании сложных структур данных, и сосредоточьте все внимание на одном — скорости! Во время игры вы должны иметь постоянный и быстрый доступ к ее данным, так что при выборе структур данных следует рассмотреть вопросы простоты доступа, размера данных и относительных размеров, доступности из кэша процессора и кэша второго уровня и т.п. Если вы не сможете обеспечить быстрое обращение к данным, вам не поможет даже процессор с частотой 1GHz.

Ниже представлены выведенные мною эмпирические правила проектирования структур данных.

- Отдавайте предпочтение простоте.
- Для небольших структур, количество которых известно заранее, лучше использовать статические массивы.
- В случае необходимости используйте связанные списки.
- Деревья и другие экзотические структуры данных используйте только тогда, когда они способствуют увеличению скорости выполнения кода, а не просто потому, что вы с ними знакомы.
- И наконец, при проектировании структур данных учитывайте возможность их расширения. Не загоняйте себя в угол использованием структур данных, которые запрещают добавление новых характеристик или накладывают излишние ограничения на объекты.

Но достаточно проповедей. Рассмотрим базовую структуру данных для хранения информации о многоугольнике. Предположим, многоугольник имеет много вершин; в таком случае для сохранения этих вершин статический массив не подходит, и вершины многоугольника следует хранить в динамическом массиве. Кроме того, вам понадобятся координаты многоугольника, его скорость (об этом — немного позже), цвет и, возможно, дополнительная информация о его состоянии. Итак, первый вариант подобной структуры выглядит следующим образом:

```
typedef struct POLYGON2D_TYP
{
  int    state;       // Состояние многоугольника
  int    num_verts;   // Число вершин
  int    x0,y0;       // Координаты многоугольника
  int    xv, yv;      // Скорость
```

```
  DWORD     color;      // Индекс или PALETTENTRY
  VERTEX2DI *vlist;     // указатель на список вершин
} POLYGON2D, *POLYGON2D_PTR;
```

C++

Этот фрагмент программы идеально кодируется в C++ с использованием классов, но мне хотелось бы учесть интересы поклонников языка C.

Все вроде бы неплохо, но непонятно, что такое VERTEX2DI. Это обычная история при проектировании структур данных: вы что-то не определили, но знаете, что это вам понадобится. Теперь необходимо определить структуру VERTEX2DI, которая просто содержит целочисленную двухмерную вершину:

```
typedef struct VERTEX2DI_TYP
{
  int x,y;
} VERTEX2DI, *VERTEX2DI_PTR;
```

$\int\sum_{\alpha}^{\infty}$

Во многих двух- и трехмерных машинах все вершины задаются только как целые числа. Разумеется, при этом снижается точность всех преобразований масштабирования и поворота. Проблема, связанная с числами с плавающей точкой, состоит в том, что эти числа медленно преобразуются в целые. Несмотря на то что Pentium может выполнять операции с плавающей точкой так же быстро (или даже быстрее), как и операции с целыми числами, преобразование значений с плавающей точкой в целочисленные в конце процесса растеризации способно лишить вас радости победы. Подобное преобразование не составляет проблемы до тех пор, пока происходит в *самом конце* вычислений; но если эти преобразования выполняются в процессе работы программы, о высокой производительности придется забыть. Главный вывод отсюда — если точность вычислений при использовании целочисленных значений вас устраивает, работайте с целыми числами. В противном случае воспользуйтесь числами с плавающей точкой, но отложите их преобразование в целые на последний момент.

Итак, у нас есть замечательная структура данных для хранения вершин и многоугольника. На рис. 8.14 показана связь структуры данных с реальным многоугольником.

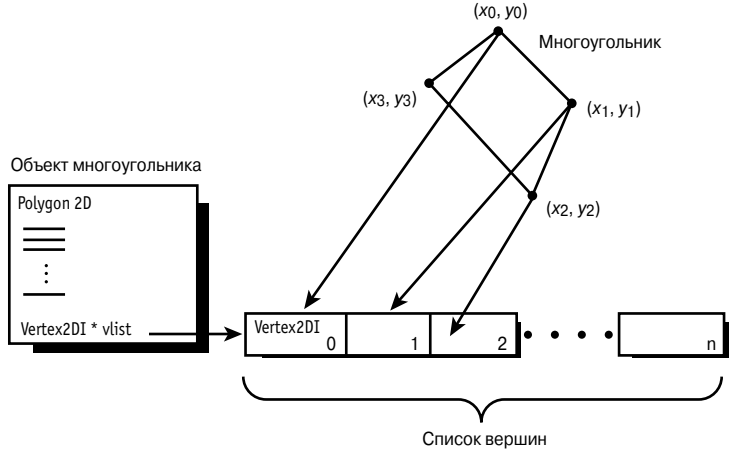


Рис. 8.14. Структура данных, предназначенная для многоугольника

Чтобы использовать структуру POLYGON, вы должны создать переменную этого типа и заполнить ее данными.

```
POLYGON2D triangle;        // Наш многоугольник

// Инициализируем треугольник
triangle.state    = 1;
triangle.num_verts = 3;    // Треугольник (три вершины)
triangle.x0       = 100;   // Положение треугольника
triangle.y0       = 100;
triangle.xv       = 0;     // Скорость
triangle.yv       = 0;
triangle.color    = 50;    // Индекс в 8-битовом режиме
triangle.vlist    = new VERTEX2DI[triangle.num_verts];
```

C++

Обратите внимание, что для выделения памяти я использовал оператор new. Поэтому для освобождения этой памяти я должен буду использовать оператор delete. В языке C подобное резервирование памяти делается с помощью функции malloc: (VERTEX2DI_PTR)malloc(triangle.num_verts*sizeof(VERTEX2DI)).

Теперь посмотрим, как можно нарисовать такие многоугольники.

Черчение и отсечение многоугольников

Черчение многоугольника представляет собой просто вычерчивание *n* связанных отрезков прямых линий. Вы уже знаете, как чертится линия, поэтому все, что вам нужно сделать для вычерчивания многоугольника, — это обойти все вершины и соединить их. Разумеется, если вам нужно отсечь многоугольник, это делается посредством вызова функции черчения линии с учетом отсечения Draw_Clip_Line(). Ее параметры ничем не отличаются от параметров функции Draw_Line(), кроме того, что она выполняет отсечение в соответствии с глобально определенной отсекающей областью. Итак, напишем общую функцию вычерчивания многоугольника, описываемого структурой POLYGON2D:

```
int Draw_Polygon2D(POLYGON2D_PTR poly,
           UCHAR *vbuffer, int lpitch)
{
  // Сначала проверяем, является ли многоугольник
  // POLYGON2D видимым, а затем выводим его
  if (poly->state)
  {
    // Проходим по всем вершинам от 1 до n
    // и вычерчиваем линии
    for(int index=0; index<poly->num_verts-1; index++)
    {
      // Чертим линию, соединяющую вершины i и i+1
      Draw_Clip_Line(poly->vlist[index].x+poly->x0,
                     poly->vlist[index].y+poly->y0,
                     poly->vlist[index+1].x+poly->x0,
                     poly->vlist[index+1].y+poly->y0,
                     poly->color,
                     vbuffer, lpitch};
           } // for
    // Замкнем многоугольник - начертим линию,
    // соединяющую последнюю вершину с первой
    Draw_Clip_Line(poly->vlist[0].x+poly->x0,
              poly->vlist[0].y+poly->y0,
```

```
                poly->vlist[index].x+poly->x0,
                poly->vlist[index].y+poly->y0,
                poly->color,
                vbuffer, lpitch};

// Возвращаем код успешного завершения функции
return (1)
} // if
else
   return (0);

} // Draw_Polygon2D
```

В этой функции особого внимания заслуживает одна вещь — использование координат (x_0,y_0) в качестве центра многоугольника, что позволяет избежать путаницы с вершинами при его перемещении. Более того, определяя многоугольник относительно его центра, можно использовать не *внешние (мировые) координаты*, а так называемые *локальные координаты*. Взаимосвязь между этими двумя типами координат представлена на рис. 8.15.

Рис. 8.15. Связь между локальными и мировыми координатами

Локальные координаты связаны с внешними координатами. Всегда лучше определять многоугольники относительно центра $(0,0)$ (в локальных координатах), а затем преобразовывать координаты многоугольника в (x,y) (мировые координаты). Пока же просто отметим, что такие координаты существуют.

В качестве демонстрационного примера я создал программу DEMO8_3.CPP, которая создает массив восьмиугольных объектов, похожих на маленькие астероиды. Затем программа случайным образом располагает эти астероиды и перемещает их по экрану. Программа работает в режиме разрешения экрана 640×480×8 и для осуществления анимации использует переключение страниц. На рис. 8.16 показана копия экрана этой программы.

Теперь вы умеете определять многоугольник и рисовать его. И мне пора перейти к теме двухмерных преобразований: переносу, повороту и масштабированию многоугольников.

Рис. 8.16. Программа DEMO8_3.EXE в действии

Преобразования, выполняемые на плоскости

Я уверен, что теперь вы уже не так боитесь математики и готовы погрузиться в ее мир. Это на самом деле очень интересно, и если вы поймете основные принципы, то в программировании трехмерных игр для вас действительно не будет ничего невозможного! Итак, приступим.

До этого момента мы сталкивались с переносом уже несколько раз, но не давали ему математического обоснования (равно как и другим преобразованиям, например повороту или масштабированию). Осталось рассмотреть каждое из этих понятий и их связь с двухмерными векторными изображениями. Тогда при переходе к трехмерной графике вам нужно будет только добавить одну-две переменные и принять во внимание ось Z.

Перенос

Это не что иное, как перемещение объекта или точки из одного места в другое. Допустим, у вас есть точка (x,y), которую вы хотите переместить на некоторое расстояние, (d_x,d_y). Процесс переноса представлен на рис. 8.17.

Для этого вы должны добавить вектор переноса к координатам (x,y) и получить новые координаты — (x_t,y_t). Математически эта операция описывается так:

$$x_t = x + d_x;$$
$$y_t = y + d_y.$$

Здесь значения d_x и d_y могут быть как положительными, так и отрицательными. Если перенос выполняется в стандартной системе координат экрана, где точка $(0,0)$ соответствует верхнему левому углу, то при положительном переносе объекта вдоль оси X он перемещается вправо, а при положительном переносе вдоль оси Y — вниз. Соответственно при отрицательном переносе вдоль оси X объект перемещается влево, а при отрицательном переносе вдоль оси Y — вверх.

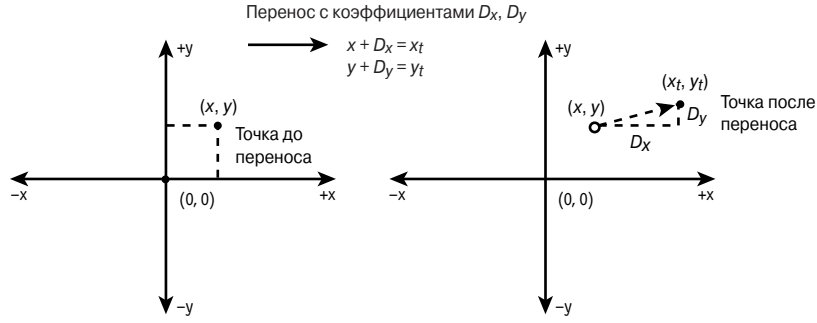


Рис. 8.17. Перенос одной точки

Чтобы перенести весь объект, вы просто применяете преобразование переноса к центру объекта, относительно которого рассчитываются все точки (как, например, в структуре, описывающей многоугольник). Если для объекта не определены общие координаты, вы должны применить формулу к каждой точке, входящей в многоугольник. Здесь мы вновь сталкиваемся с концепцией локальных и мировых координат.

В целом в компьютерной графике нужно определять все объекты, как минимум, в локальной и мировой системах координат. Локальные координаты объекта вычисляются относительно точки $(0,0)$ (или $(0,0,0)$ в трехмерном представлении). Тогда мировые координаты могут быть получены добавлением координат мировой позиции (x_0, y_0) к соответствующим локальным координатам, по сути, путем переноса каждой точки с помощью вектора перемещения (x_0, y_0). Этот процесс показан на рис. 8.18.

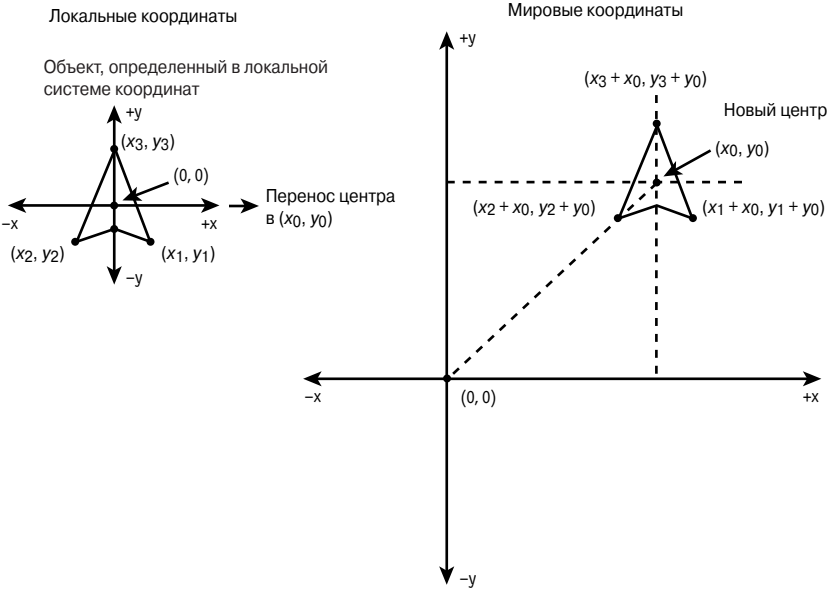


Рис. 8.18. Перенос объекта и новые координаты вершин

Исходя из сказанного, вы можете решить, что имеет смысл использовать дополнительные переменные в структуре данных многоугольника с тем, чтобы хранить в ней как локальные, так и мировые координаты (что и будет в действительности сделано позднее). Кроме того, вам

потребуются переменные для хранения координат камеры. Добавление новых переменных объясняется тем, что, если вы уже один раз выполнили преобразование координат объекта к внешним координатам и готовы к его выводу, вряд ли вы захотите делать такое преобразование в каждом отдельном кадре. Пока объект не перемещается или не переносится, вам не нужны дополнительные вычисления — вы можете просто сохранить последние вычисленные значения мировых координат объекта и пользоваться ими.

Рассмотрим общую функцию переноса многоугольников. Она на удивление проста:

```
int Translate_Polygon2D(POLYGON2D_PTR poly, int dx, int dy)
{
  // Эта функция выполняет перенос центра многоугольника

  // Проверяем правильность указателя
  if (!poly) return (0);

  // Перенос
  poly->x0 += dx;
  poly->y0 += dy;

  // Возвращаем код успешного завершения функции
  return (1);

} // Translate_Polygon2D
```

Поворот

Поворот растровых изображений достаточно сложен, но поворот одиночных точек на плоскости вполне тривиален. Перед тем, как рассмотреть осуществление поворотов, давайте выясним, что именно вы должны сделать. На рис. 8.19 изображена точка p_0 с координатами (x,y). Вам нужно повернуть эту точку на угол θ вокруг оси Z (которая перпендикулярна плоскости, в которой находятся оси X и Y) и найти новую точку p_0' с координатами (x_r,y_r).

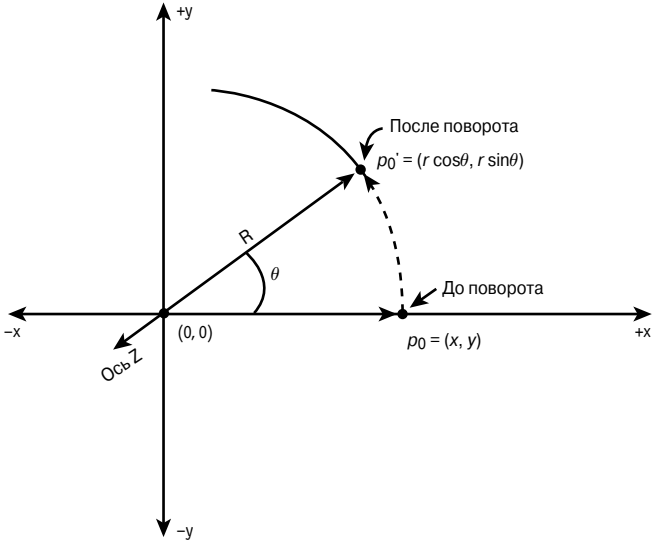


Рис. 8.19. Поворот точки

Немного тригонометрии

Сейчас придется использовать некоторые тригонометрические функции. Если вы немного их подзабыли, я вкратце напомню некоторые основные сведения.

Большая часть тригонометрии, как видно из рис. 8.20, основана на анализе прямоугольного треугольника.

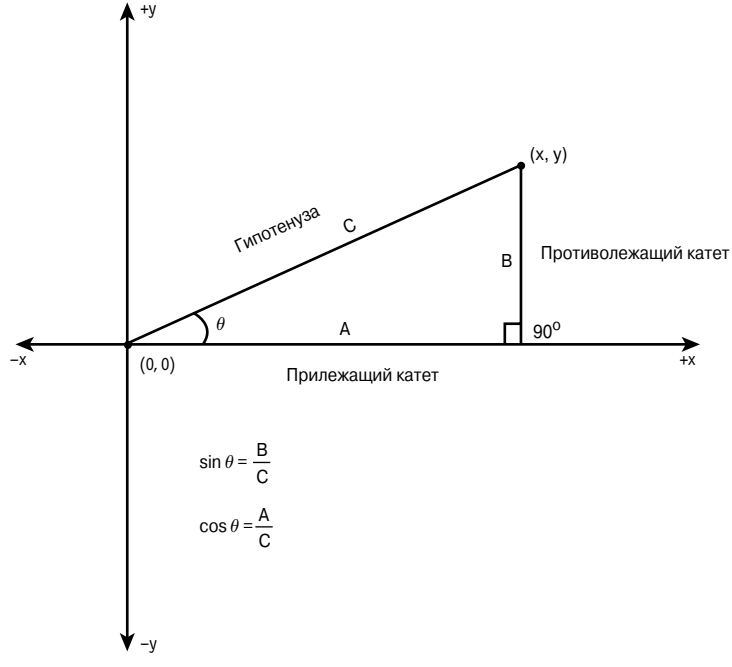


Рис. 8.20. Прямоугольный треугольник

Таблица 8.1. Радианы и градусы

Градусы	*Радианы (π)*	*Радианы (численное значение)*
360	2π	6.28
180	π	3.14159
90	$\pi/2$	1.57
57.295	1	1.0
1	$\pi/180$	0.0175

Ниже перечислены некоторые тригонометрические факты.

- Полная окружность содержит 360°, или 2π радиан (соответственно 180° составляют π радиан). Функции sin() и cos() из библиотеки C используют значение угла, заданное в радианах, а *не* в градусах, — об этом следует помнить! В табл. 8.1 представлены некоторые значения углов в градусах и соответствующие им значения в радианах.
- Сумма внутренних углов треугольника равна 180°, или π радиан: $\theta_1 + \theta_2 + \theta_3 = \pi$.
- В прямоугольном треугольнике, изображенном на рис. 8.20, сторона, лежащая напротив угла θ, называется *противолежащей*, нижняя сторона называется *прилежащей*, а длинная сторона прямоугольного треугольника называется *гипотенузой*.

- Сумма квадратов катетов прямоугольного треугольника равна квадрату гипотенузы. Это утверждение называется *теоремой Пифагора*. Поэтому, если известны две стороны прямоугольного треугольника, всегда можно найти третью сторону.
- Математики чаще всего используют три основных тригонометрических соотношения: *синус*, *косинус* и *тангенс*. Они определяются следующим образом:

$$\sin\theta = \frac{y}{r}, 0 \le \theta \le 2\pi, -1 \le \sin \le 1;$$

$$\cos\theta = \frac{x}{r}, 0 \le \theta \le 2\pi, -1 \le \cos \le 1;$$

$$\mathrm{tg}\,\theta = \frac{\sin\theta}{\cos\theta} = \frac{y}{x}, -\frac{\pi}{2} \le \theta \le \frac{\pi}{2}, -\infty < \mathrm{tg} < +\infty.$$

На рис. 8.21 представлены графики всех этих функций. Следует заметить, что все они являются периодическими; период синуса и косинуса равен 2π, а период тангенса равен π. Заметим также, что тангенс стремится к бесконечности, когда значение θ по модулю π равно $\pi/2$.

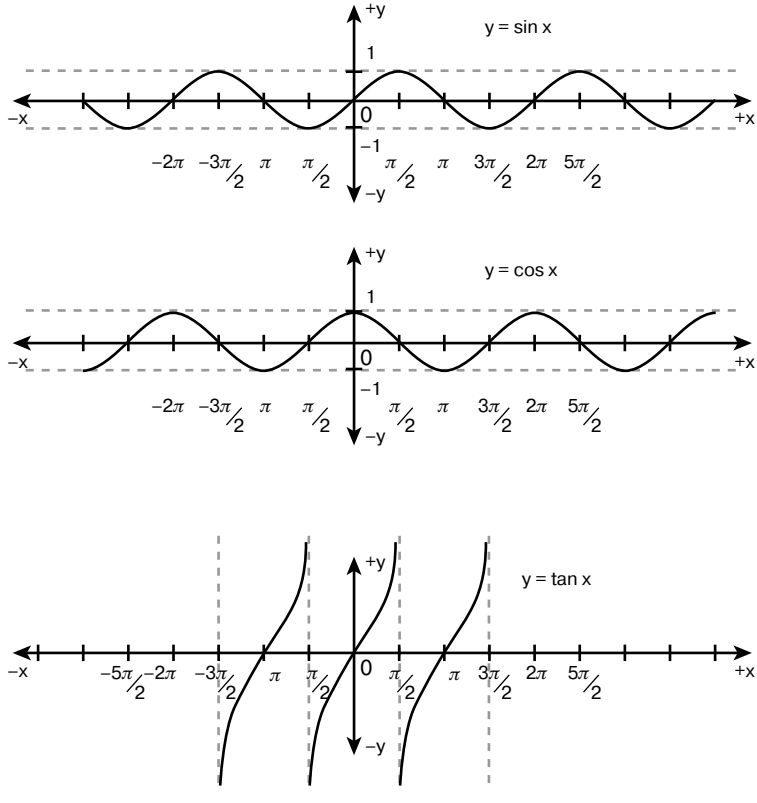


Рис. 8.21. Графики основных тригонометрических функций

Теперь кратко рассмотрим свойства тригонометрических функций. Хотя об этом можно написать не одну книгу, я просто хочу представить вам ряд свойств, которые могут пригодиться при программировании игр. В табл. 8.2 показаны некоторые тригонометрические отношения и тождества.

Таблица 8.2. Полезные тригонометрические соотношения и тождества

Косеканс: $\operatorname{cosec}\theta = 1/\sin\theta$

Секанс: $\sec\theta = 1/\cos\theta$

Котангенс: $\operatorname{ctg}\theta = 1/\operatorname{tg}\theta$

Теорема Пифагора, выраженная через тригонометрические функции:

$$(\sin\theta)^2 + (\cos\theta)^2 = 1$$

Тригонометрические преобразования:

$$\sin\theta = \cos(\theta - \pi/2)$$

$$\sin(-\theta) = -\sin(\theta)$$

$$\cos(-\theta) = \cos(\theta)$$

$$\sin(\theta \pm \phi) = \sin\theta\cos\phi \pm \cos\theta\sin\phi$$

$$\cos(\theta \pm \phi) = \cos\theta\cos\phi \mp \sin\theta\sin\phi$$

Всегда, когда вы сталкиваетесь с алгоритмом, использующим тригонометрические функции, не поленитесь заглянуть в справочник и выяснить, нельзя ли упростить математические выражения и уменьшить количество вычислений, необходимых для достижения результата. Не забывайте — скорость, скорость и еще раз скорость!!!

Поворот точки на плоскости

Теперь, когда я напомнил вам о тригонометрических функциях, осталось выяснить, как они используются для вычисления поворота точки, расположенной на двухмерной плоскости. Рис. 8.22 позволяет понять, как выводятся формулы поворота.

Начнем с того, что координаты любой точки на окружности радиуса R с центром в начале координат могут быть вычислены при помощи следующих формул:

$$x_r = R\cos\theta,$$
$$y_r = R\sin\theta.$$

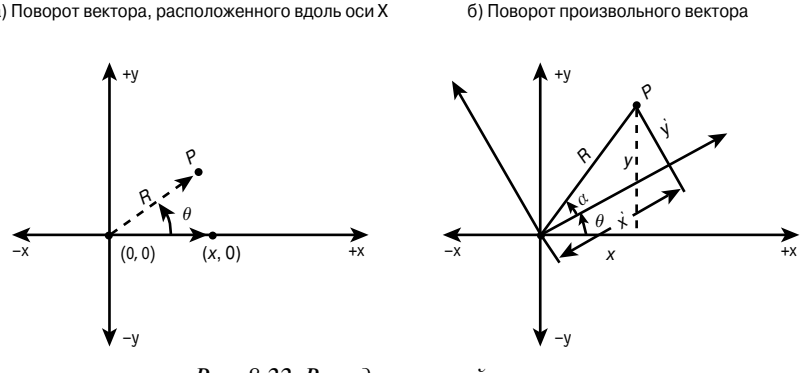


Рис. 8.22. Вывод уравнений поворота

Следовательно, эти формулы можно использовать при повороте точки с координатами $(x,0)$. Поворот точки (x,y) на угол θ (показанный на рис. 8.22) можно рассматривать и как поворот точки P, и как поворот самих осей координат. Если рассмотреть поворот осей координат, у нас окажется две системы координат: одна — до вращения, другая — после.

После поворота на угол α в исходной системе координат справедливы следующие формулы:

$$x_r = r\cos(\theta+\alpha),$$
$$y_r = r\sin(\theta+\alpha),$$

а в повернутой системе координат — формулы

$$x = r\cos\alpha,$$
$$y = r\sin\alpha.$$

Здесь α — это угол, образованный осью X новой системы координат и радиусом-вектором, начинающимся в точке начала координат и заканчивающимся в точке P.

Если вы запутались, попробую объяснить по-другому. Всегда известно, как найти точку $(x,0)$, повернутую на угол θ, и если вы поворачиваете оси координат на тот же угол, то можете вычислить P и в старой системе координат, и в новой. Затем, исходя из этих двух формул, вы получаете уравнения поворота. Если взять первые уравнения и применить к ним формулы сложения тригонометрических функций, получим:

$$x_r = r\cos\theta\cos\alpha - r\sin\theta\sin\alpha,$$
$$y_r = r\sin\theta\cos\alpha + r\cos\theta\sin\alpha.$$

Подставляя сюда значения для $\sin\alpha$ и $\cos\alpha$ из уравнений для повернутой системы координат, получим:

$$x_r = x\cos\theta - y\sin\theta,$$
$$y_r = x\sin\theta + y\cos\theta.$$

С математической точки зрения этот вывод уравнения очень похож на поворот в полярной системе координат с последующим преобразованием в декартову. Именно так я его и получил.

Вернемся к реальности. Теперь нам известно, какие формулы следует использовать для поворота точки (x,y) на угол θ. Здесь необходимо отметить одну важную деталь: если угол θ положителен, поворот выполняется против часовой стрелки, а если отрицателен — по часовой стрелке. Однако уравнение было выведено для обычной прямоугольной системы координат, так что на экране дисплея ось Y оказывается инвертированной, а значит, положительный и отрицательный углы меняются местами.

Поворот многоугольника

Учитывая все изложенное, напишем функцию поворота, которая вращает многоугольник.

```
int Rotate_Polygon2D(POLYGON2D_PTR poly, float theta)
{
  // Функция выполняет поворот локальных
  // координат многоугольника

  // Проверяем корректность указателя
  if (!poly)
    return (0);
```

```
  // Цикл поворота всех точек (в лоб, без
  // использования таблиц!)
  for(int curr_vert = 0; curr_vert < poly->num_verts;
    curr_vert++)
  {
    float xr = poly->vlist [curr_vert].x*cos(theta) -
          poly->vlist [curr_vert].y*sin(theta);
    float yr = poly->vlist [curr_vert].x*sin(theta)+
          poly->vlist [curr_vert].y*cos(theta);

    // Сохраняем результаты в тех же переменных
    poly->vlist [curr_vert].x = xr;
    poly->vlist [curr_vert].y = yr;

  } // for

  // Возвращаем код успешного завершения функции
  return (1);

} // Rotate_Polygon2D
```

Здесь следует отметить сразу несколько моментов. Во-первых, математические операции выполняются в формате с плавающей точкой, а затем результаты сохраняются как целые числа, поэтому налицо потеря точности.

Во-вторых, функция получает в качестве параметра значение угла в радианах (а не в градусах), так как использует библиотечные функции sin() и cos(), в которых значение угла задается в радианах. Потеря точности не представляет большой проблемы, но использование тригонометрических функций в программах реального времени — это самое ужасное, что только может быть. Вам нужно заранее создать таблицу, содержащую значения синусов и косинусов для углов, например от 0 до 360°, а затем заменить обращения к библиотечным функциям sin() и cos() поиском в таблице.

Как же создать такую таблицу? Решение этой задачи зависит от конкретной ситуации. Некоторым, вероятно, захочется использовать для индексации таблицы один байт, и окружность при этом можно представить как состоящую из 256 виртуальных градусов (рис. 8.23).

Я же обычно предпочитаю, чтобы таблица содержала углы от 0 до 359°. Подобную таблицу можно создать следующим образом:

```
// Массивы для хранения таблиц
float cos_look[360];
float sin_look[360];

// Создаем таблицу
for(int ang = 0; ang < 360; ang++)
{
  // Преобразуем значение угла в радианы
  float theta = (float)ang * 3.14159 / 180;

  // Помещаем в таблицу очередной элемент
  cos_look[ang] = cos[theta];
  sin_look[ang] = sin[theta];

} // for
```

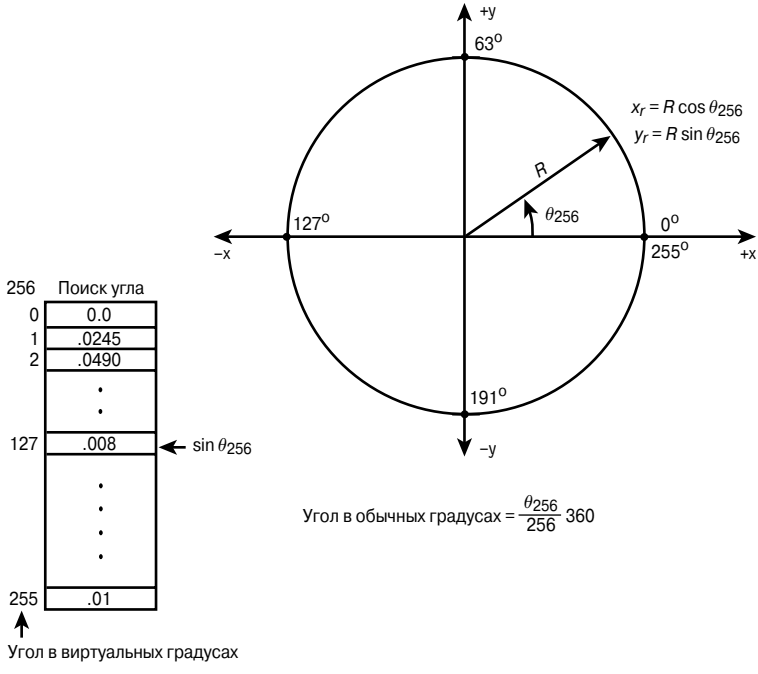


Рис. 8.23. Разбиение окружности на 256 виртуальных градусов

Затем можно переписать функцию поворота и заставить ее принимать в качестве параметра значения углов от 0 до 359° и использовать табличные значения, заменяя функции sin() и cos() обращениями к элементам массивов sin_look[] и cos_look[] соответственно.

```
int Rotate_Polygon2D(POLYGON2D_PTR poly, int theta)
{
  // Функция выполняет поворот локальных
  // координат многоугольника

  // Проверяем корректность указателя
  if (!poly)
    return (0);

  // Цикл поворота всех точек
  for(int curr_vert = 0; curr_vert < poly->num_verts;
    curr_vert++)
  {
    float xr = poly->vlist [curr_vert].x*cos_look[theta] -
          poly->vlist [curr_vert].y*sin_look[theta];
    float yr = poly->vlist [curr_vert].x*sin_look[theta]+
          poly->vlist [curr_vert].y*cos_look[theta];

    // Сохраняем результаты в тех же переменных
    poly->vlist [curr_vert].x = xr;
    poly->vlist [curr_vert].y = yr;

  } // for
```

```
  // Возвращаем код успешного завершения функции
  return (1);

} // Rotate_Polygon2D
```

Чтобы повернуть объект типа POLYGON2D на 10°, необходимо при вызове функции передать ей следующие параметры:

```
Rotate_Polygon2D(&object, 10);
```

НА ЗАМЕТКУ

Замечу, что такой способ поворота приводит к искажению первоначальных координат многоугольника. Конечно, если вы поворачиваете многоугольник на 10°, то всегда можете повернуть его и на -10°, но при этом координаты исходных вершин будут постепенно изменяться из-за округления и усечения целочисленных значений. Именно поэтому лучше иметь второй набор координат, который сохраняется в описывающей многоугольник структуре. Хранение оригиналов координат в случае необходимости позволяет обновить данные.

Немного о точности

Вначале я написал демонстрационную программу с сохранением вершин в целочисленных переменных, но обнаружил, что вершины "уплывают" уже после нескольких операций поворота. Поэтому мне пришлось переписать программу с использованием чисел с плавающей точкой для хранения вершин. Вы должны переопределить свою структуру POLYGON2D так, чтобы содержащиеся в ней координаты вершин задавались не целочисленными значениями, а числами с плавающей точкой.

Существует два способа решения этой задачи. Один заключается в том, чтобы иметь набор локальных и преобразованных координат (оба набора могут быть заданы как целые числа) и в каждом кадре для вычислений использовать исходные значения координат. Так можно избавиться от ошибки в значениях локальных координат.

Второй способ предполагает хранение одного набора локальных координат в формате с плавающей точкой (я поступил именно так). В результате мы используем следующие структуры данных для вершин и для многоугольника:

```
// Двухмерная вершина
typedef struct VERTEX2DF_TYP
{
  float x,y; // Координаты вершины
} VERTEX2DF, *VERTEX2DF_PTR;

// Плоский многоугольник
typedef struct POLYGON2D_TYP
{
  int      state;          // Состояние многоугольника
  int      num_verts;      // Число вершин
  int      x0,y0;          // Координаты центра многоугольника
  int      xv, yv;         // Начальная скорость
  DWORD    color;          // Индекс цвета или PALETTENTRY
  VERTEX2DF *vlist;        // Указатель на список вершин
} POLYGON2D,  *POLYGON2D_PTR;
```

Я просто заменил список вершин новыми значениями с плавающей точкой, поэтому мне не пришлось переписывать все и вся. Теперь будут корректно работать и перенос и поворот, хотя при переносе все еще используются целые числа. Безусловно, я бы мог сделать это и раньше, но мне хотелось показать весь процесс, саму суть программирова-

ния игр. Вы надеетесь, что все в порядке, но когда происходит сбой, вы возвращаетесь на шаг назад и делаете еще одну попытку.

Чтобы показать использование обеих функций — поворота и переноса, я переписал демонстрационную программу DEMO8_3.CPP и назвал ее DEMO8_4.CPP. Эта программа поворачивает все астероиды с различной скоростью и использует таблицы поиска. Обязательно посмотрите, как она работает.

Масштабирование

После этого материала любые другие операции не должны вызвать у вас каких-либо затруднений. Масштабирование выполняется почти так же просто, как и перенос. Рассмотрим рис. 8.24. При масштабировании объекта каждая координата просто умножается на соответствующий множитель.

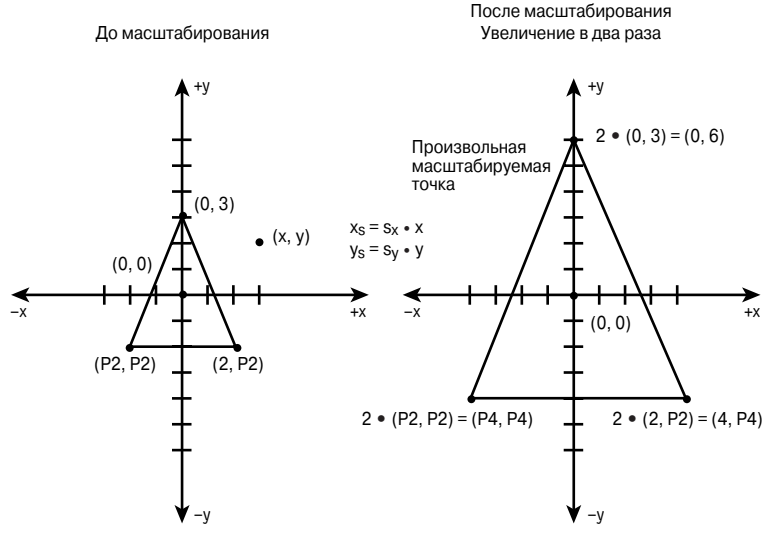


Рис. 8.24. Математика масштабирования

Если значение масштабирующего множителя больше 1.0, объект будет увеличен, если меньше 1.0 — уменьшен. Если масштабирующий множитель равен 1.0, с объектом ничего не происходит. Математические уравнения, описывающие масштабирование точки (x,y) при помощи множителя s для получения результирующей точки (x_s,y_s), имеют вид

$$x_s = xs,$$
$$y_s = ys.$$

Кроме того, объект можно масштабировать неоднородно, по-разному для каждой оси; при этом координаты X и Y умножаются на разные масштабирующие множители:

$$x_s = xs_x,$$
$$y_s = ys_y.$$

Как правило, вам придется выполнять однородное масштабирование. Тем не менее, кто знает, может, вам потребуется увеличение объекта только вдоль одной из осей? Вот функция, которая масштабирует многоугольник с разными коэффициентами для разных координат:

```
int Scale_Popygon2D(POLYGON2D_PTR poly, float sx, float sy)
{
```

```
  // Функция масштабирует локальные координаты
  // многоугольника

  // Проверяем корректность указателя
  if (!poly)
     return (0);

  // Масштабируем каждую точку
  for(int curr_vert = 0; curr_vert < poly->num_verts;
     curr_vert++)
  {
     // Выполняем масштабирование и сохраняем результаты
     //  в тех же переменных
     poly->vlist[curr_vert].x *= sx;
     poly->vlist[curr_vert].y *= sy;

  } // for

  // Возвращаем код успешного завершения
  return (1);

} // Scale_Polygon2D
```

Здесь нет ничего сложного, не правда ли?

Чтобы уменьшить многоугольник в 10 раз, нужно вызвать эту функцию со следующими параметрами:

```
Scale_Polygon2D (&polygon, 0.1, 0.1)
```

Обратите внимание, что в данном вызове функции оба коэффициента масштабирования (и по оси X, и по оси Y) равны 0.1, поэтому масштабирование является однородным, т.е. одинаковым вдоль обеих осей.

Я написал демонстрационную версию программы масштабирования DEMO8_5.CPP. Она создает один поворачивающийся астероид; при нажатии клавиши <A> объект увеличивается на 10%, а при нажатии клавиши <S> — уменьшается на 10%.

НА ЗАМЕТКУ

Вы могли заметить, что в большинстве демонстрационных программ указатель мыши остается видимым. Если вы хотите убрать его (что совсем неплохо для игр), можете обратиться к функции Win32 ShowCursor(BOOL bshow). Если передаваемый ей параметр имеет значение TRUE, внутренний счетчик экрана увеличится на 1, если FALSE — уменьшится на 1. При запуске системы счетчик экрана равен нулю. При значении счетчика экрана, большем или равном нулю, указатель мыши видим, поэтому первый вызов функции ShowCursor(FALSE) сделает его невидимым. Помните, что в счетчике функции ShowCursor() происходит накопление всех обращений к ней, поэтому, если функция пять раз вызывалась с параметром FALSE, нужно пять раз вызвать ее с параметром TRUE, чтобы "открутить" показания счетчика в обратном направлении.

Введение в матрицы

Сейчас я просто хочу описать некоторые свойства матриц и их использование в простых двухмерных преобразованиях, которые вам придется выполнять очень часто, но при работе с трехмерной графикой обойтись без матриц просто невозможно.

Матрица представляет собой не что иное, как прямоугольный массив чисел с заданным количеством строк и столбцов. Считается, что матрица имеет *размерность* $m \times n$, если у нее m строк и n столбцов. Приведем пример матрицы A размерностью 2×2.

$$A=\begin{bmatrix}1 & 4\\ 9 & -1\end{bmatrix}.$$

Заметим, что для обозначения данной матрицы я использовал прописную букву *A*. Как правило, матрицы обозначаются прописными буквами, а векторы — буквами, выделенными жирным шрифтом. В указанном примере первая строка — это <1 4>, а вторая — <9 -1>. Вот примеры матриц 3×2 и 2×3:

$$B=\begin{bmatrix}5 & 6\\ 2 & 3\\ 100 & -7\end{bmatrix},\ C=\begin{bmatrix}3 & 5 & 0\\ -8 & 12 & 4\end{bmatrix}.$$

Чтобы получить <*i*,*j*>-й элемент матрицы, нужно найти значение, находящееся в *i*-й строке и *j*-м столбце. Однако почти во всех книгах по математике нумерация элементов матрицы начинается не с 0 (как в компьютерных программах), а с 1, и это следует иметь в виду. Мы будем начинать нумерацию элементов матриц с нуля, так как в этом случае работа матриц, используемых в программах на языке C/C++, выглядит более естественно. Приведем пример обозначения элементов матрицы размерностью 3×3:

$$A=\begin{bmatrix}a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{bmatrix}.$$

Достаточно просто. Пока это все, что касается непосредственно самих матриц и принятых обозначений. Но вы можете спросить, зачем нам эти матрицы? Они просто являются математическим инструментом для ленивых математиков (шучу, шучу...). Как правило, если у вас есть некоторая система уравнений, например

$$\begin{cases}3x+2y=1,\\ 4x-9y=9.\end{cases}$$

то для записи всех переменных потребуется немало усилий. Известно, что это переменные (x,y), так почему нужно обязательно повторять их написание несколько раз? Почему бы просто не создать компактный формат записи, содержащий те данные, с которыми вам предстоит работать, только по одному разу? Именно так и появились матрицы. В предыдущем примере имеется три разных набора значений, которые можно представить с помощью матриц. Эти значения могут рассматриваться как все вместе, так и каждое в отдельности.

Коэффициенты приведенной системы уравнений записываются в виде матрицы

$$A=\begin{bmatrix}3 & 2\\ 4 & -9\end{bmatrix},$$

размерность которой равна 2×2. Переменные этой системы уравнений записываются в виде матрицы

$$X=\begin{bmatrix}x\\ y\end{bmatrix},$$

размерность которой равна 2×1, а константы правой части системы уравнений также можно представить матрицей 2×1:

$$B=\begin{bmatrix}1\\ 9\end{bmatrix}.$$

Вы можете работать с любой из этих матриц, например с матрицей *A*, содержащей коэффициенты, не заботясь обо всех остальных данных. Сама система уравнений может быть записана в матричном виде как

$$AX = B.$$

Если выполнить соответствующие математические операции, мы получим исходную систему уравнений

$$\begin{cases} 3x + 2y = 1, \\ 4x - 9y = 9. \end{cases}$$

Но как выполнить такое математическое преобразование? Этим мы сейчас и займемся.

Единичная матрица

В любой математической системе прежде всего необходимо дать определение 1 и 0. В матричной математике тоже существуют аналоги этих величин. Аналог единицы называется *единичной матрицей*, которая получается посредством задания единиц для всех элементов, расположенных вдоль главной диагонали матрицы, и нулей — для всех остальных элементов. Поскольку матрицы могут иметь любые размеры, очевидно, что число единичных матриц бесконечно. Но здесь есть одно ограничение: все единичные матрицы должны быть квадратными, иными словами, иметь размерность $m \times m$, где $m \geq 1$. Приведем два примера единичных матриц размерностью 2×2 и 3×3:

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \; I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Строго говоря, единичная матрица не является точным аналогом единицы, но является таковым в контексте операции умножения матриц (которое будет рассмотрено ниже). Ко второму типу фундаментальных матриц относится *нулевая матрица*, которая является нулевой в контексте операций сложения и умножения. Она представляет собой не что иное, как матрицу размерностью $m \times n$, все элементы которой равны 0. Других специальных ограничений для этой матрицы не существует.

$$Z_{3\times3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \; Z_{1\times2} = \begin{bmatrix} 0 & 0 \end{bmatrix}.$$

Единственная замечательная черта нулевой матрицы заключается в том, что она обнаруживает стандартные свойства скалярного нуля как при операции сложения, так и при операции умножения. На этом вся ее привлекательность заканчивается.

Сложение матриц

Сложение и вычитание матриц выполняется путем сложения или вычитания всех соответствующих элементов обеих матриц и получения элементов результирующей матрицы. Единственное правило, касающееся вычитания и сложения, гласит, что участвующие в операции сложения матрицы должны иметь одинаковую размерность.

$$A = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix}, \; B = \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix},$$

$$A + B = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} + \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix} = \begin{bmatrix} 14 & 12 \\ 3 & -10 \end{bmatrix},$$

$$A-B=\begin{bmatrix}1 & 5\\ -2 & 0\end{bmatrix}-\begin{bmatrix}13 & 7\\ 5 & -10\end{bmatrix}=\begin{bmatrix}-12 & -2\\ -7 & 10\end{bmatrix}.$$

Заметим, что обе операции — и сложение и вычитание — являются ассоциативными, поэтому $A+(B+C)=(A+B)+C$. Очевидно, что операция вычитания не коммутативна (т.е. $A-B\neq B-A$).

Умножение матриц

Существует два вида операции умножения матриц: *скалярное* и *матричное*. Скалярное умножение представляет собой простое умножение матрицы на скалярное значение, при этом каждый элемент матрицы просто умножается на это число. Размерность матрицы $m\times n$ может быть произвольной, например:

$$A=\begin{bmatrix}a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{bmatrix},\ kA=\begin{bmatrix}ka_{00} & ka_{01} & ka_{02}\\ ka_{10} & ka_{11} & ka_{12}\\ ka_{20} & ka_{21} & ka_{22}\end{bmatrix},$$

$$3*\begin{bmatrix}1 & 4\\ -2 & 6\end{bmatrix}=\begin{bmatrix}3 & 12\\ -6 & 18\end{bmatrix}.$$

Умножение второго вида — это матричное умножение. Его математическая форма выглядит немного сложнее. Мы рассматриваем матрицу как некоторый "оператор", воздействующий на другую матрицу. Пусть заданы две перемножаемые матрицы — A и B (они должны иметь одинаковые внутренние размеры. Иными словами, если размерность матрицы A — $m\times n$, то размерность матрицы B должна быть $n\times r$). Величины m и r могут быть любыми. Например, можно умножать матрицу 2×2 на матрицу 2×2, матрицу 3×2 на матрицу 2×3 или матрицу 4×4 на 4×5, но нельзя перемножать, например, матрицы 3×3 и 2×4, так как их внутренние размерности не равны друг другу. Результирующая матрица будет иметь размерность, образованную внешними размерами матрицы-множителя и матрицы-множимого. Например, в результате умножения матриц с размерами 2×3 и 3×4 получим матрицу 2×4.

Умножение матриц очень сложно описать словами. При подобных объяснениях мне всегда приходится прибегать к жестикуляции, поэтому лучше рассмотрим рис. 8.25, где показано, как осуществляется умножение матриц.

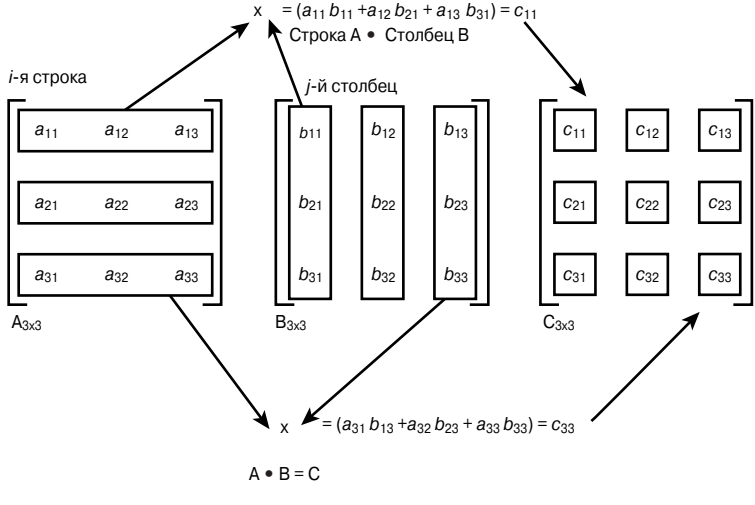


Рис. 8.25. Схема, поясняющая умножение матриц

Пусть даны матрицы *A* и *B*. Чтобы умножить эти матрицы и вычислить все элементы результирующей матрицы *C*, нужно взять строку матрицы *A* и умножить ее на столбец матрицы *B*. При умножении строки и столбца суммируются произведения их элементов. Такие произведения называются также *скалярными произведениями*. Приведем пример умножения матрицы 2×2 на матрицу 2×3 (то, что порядок умножения матриц имеет значение, очевидно хотя бы из того, что внутренние размерности должны быть одинаковы).

$$A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix},\ B=\begin{bmatrix}1 & 3 & 5\\ 6 & 0 & 4\end{bmatrix};$$

$$C=AB=\begin{bmatrix}1*1+2*6 & 1*3+2*0 & 1*5+2*4\\ 3*1+4*6 & 3*3+4*0 & 3*5+4*4\end{bmatrix}=\begin{bmatrix}13 & 3 & 13\\ 27 & 9 & 31\end{bmatrix}.$$

В качестве отступления хочу обратить ваше внимание на суммы произведений, которые представляют собой скалярные произведения векторов (*вектор* — это просто набор значений, похожий на матрицу, имеющую одну строку). Скалярное произведение имеет вполне определенный математический смысл, который мы рассмотрим ниже, но в общем случае скалярное произведение двух векторов вычисляется путем суммирования произведений его отдельных составляющих, например:

$$\mathbf{a}=\begin{bmatrix}1 & 2 & 3\end{bmatrix},\ \mathbf{b}=\begin{bmatrix}4 & 5 & 6\end{bmatrix},$$
$$\mathbf{a}\cdot\mathbf{b}=1*4+2*5+3*6=32\ .$$

Иными словами, это просто скалярная величина.

ВНИМАНИЕ

Формально скалярные произведения применимы только к векторам, но столбец или строка матрицы, по сути, представляет собой вектор.

Именно так умножаются матрицы. Можно записать это правило предельно формально:

$$A=\left[a_{ij}\right],\ B=\left[b_{jk}\right],\ AB=\left[\sum_j a_{ij}b_{jk}\right].$$

Я думаю, что теперь вам в принципе понятно, что происходит при умножении матриц. Рассмотрим фрагмент кода, реализующий его. Прежде всего создадим тип матрицы:

```
// Массив размерности 3x3
typedef struct MATRIX3X3_TYP
{
  float M[3][3];  // Массив для хранения данных
}  MATRIX3X3,  *MATRIX3X3_PTR

int Mat_Mul3X3(MATRIX3X3_PTR ma,
          MATRIX3X3_PTR mb,
          MATRIX3X3_PTR mprod)
{
  // Умножение двух матриц
  for(int row = 0; row < 3; row++)
  {
    for(int col =0; col < 3; col++)
    {
      // Вычисляем скалярное произведение строки ma
      // и столбца mb
      float sum = 0; // Результат
      for(int index=0; index<3; index++)
```

```
        {
          // Произведение очередной пары элементов
          sum += (ma->M[row][index]*mb->M[index][col];
        } // for index

        // Сохранение результата
        mprod->M[row][col] = sum;

      } // for col
    } // for row

    return (1);
} // Mat_Mul_3X3
```

Очевидно, что в этой функции вычисляется большое число математических операций. В общем случае при умножении матриц необходимо сделать порядка N^3 операций из-за наличия трех вложенных циклов. Однако в некоторых случаях можно применить оптимизацию, например проверку множителя или множимого на ноль, когда умножение не выполняется.

Преобразования, выполняемые с помощью матриц

Использование матриц в двух- и трехмерных преобразованиях нельзя назвать слишком сложным. Все, что нужно сделать, — это, как правило, умножить преобразуемую точку на некоторую матрицу преобразования. Или математически:

$$p' = pM \ ,$$

где p' — это преобразованная точка, p — исходная точка, M — матрица преобразования. Я еще не упоминал, что операция умножения матриц *не* является коммутативной, так что самое время сделать это. Итак, в общем случае $AB \neq BA$, т.е. при умножении матриц имеет значение порядок умножения.

В данном случае вы должны представить точку (x,y) в виде матрицы-строки размерностью 1×3, а затем умножить ее на матрицу преобразования размерностью 3×3. В результате получается матрица-строка размерностью 1×3, первые два элемента которой можно рассматривать как преобразованные значения координат (x',y'). Однако здесь мы сталкиваемся с небольшой проблемой: для чего нужен последний элемент первоначальной матрицы p, если используются только две координаты — x и y?

Мы представляем любую точку как вектор $\begin{bmatrix} x & y & 1 \end{bmatrix}$. Коэффициент 1 служит для превращения матрицы в так называемые *однородные координаты*, в результате чего любая преобразованная точка может быть масштабирована или перенесена. Никакого другого математического смысла здесь нет. Просто рассматривайте этот элемент как некоторую необходимую фиктивную переменную. Итак, для сохранения входной точки нужно создать матрицу 1×3, а затем умножить ее на матрицу преобразования. Используем следующую структуру данных для точки, или матрицы 1×3

```
typedef struct MATRIX1X3_TYP
{
  float M[3]; // Массив для данных точки
} MATRIX1X3, *MATRIX1X3_PTR
```

и напишем функцию для умножения координат точки на матрицу размерностью 3×3:

```
int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma,
                    MATRIX3X3_PTR mb,
                    MATRIX1X3_PTR mprod )
```

```
{
  // Эта функция умножает матрицу 1x3 на матрицу 3x3
  for(int col = 0; col < 3; col++)
  {
    // Вычисляем скалярное произведение строки ma
    // и столбца mb
    float sum = 0; // Результат произведения

    for(int index = 0; index < 3; index ++ )
    {
      // Добавляем следующее произведение
      sum += (ma->M[index]*mb->M[index][col]);
    } // for index

    // Сохраняем результат
    mprod->M[col] = sum;
  }  // for col
  return (1);

} // Mat_Mul_1X3_3X3
```

Для создания точки p с координатами (x,y) нужно сделать следующее:

```
MATRIX1X3 p = {x, y, 1};
```

Рассмотрим теперь матрицы различных преобразований.

Перенос

Чтобы осуществить перенос, к имеющимся значениям координат (x,y) нужно добавить коэффициенты переноса вдоль осей X и Y соответственно. Эту работу выполняет следующая матрица:

$$M_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{bmatrix},$$

$$p = \begin{bmatrix} x & y & 1 \end{bmatrix},\ p' = pM_t = \begin{bmatrix} x + d_x & y + d_y & 1 \end{bmatrix}.$$

Следует отметить необходимость наличия элемента 1.0 в левой матрице, представляющей точку. Без него преобразование было бы невозможно.

Итак, для первых двух элементов получаем $x' = x+d_x$ и $y' = y+d_y$, т.е. именно те преобразования, которые нам нужны.

Масштабирование

Для масштабирования точки нужно умножить координаты x и y на коэффициенты масштабирования: s_x и s_y соответственно. С учетом предположения, что во время операции масштабирования переноса нет, матрица масштабирования будет иметь следующий вид:

$$M_s = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$p = \begin{bmatrix} x & y & 1 \end{bmatrix},\ p' = pM_s = \begin{bmatrix} xs_x & ys_y & 1 \end{bmatrix},$$

т.е. мы получаем искомый результат операции масштабирования: $x' = s_x{*}x$, $y' = s_y{*}y$.

Замечание по поводу единицы в правом нижнем углу матрицы преобразования. С формальной точки зрения в ней нет необходимости, так как вам никогда не придется использовать результат вычислений, полученный для третьего столбца. Поэтому, казалось бы, в результате просто должны выполняться лишние циклы математических операций. Вопрос в том, можно ли удалить последний столбец во всех матрицах преобразования и вместо этой матрицы использовать другую — размерностью 3×2 ? Сначала рассмотрим матрицу поворота, а затем попытаемся найти ответ на этот вопрос.

Поворот

Матрица поворота является самой сложной среди всех матриц преобразований, поскольку включает в себя тригонометрические функции. В принципе у вас уже достаточно знаний, чтобы самостоятельно вывести вид матрицы поворота. Для проверки, правильно ли вы это сделали (а также для особо непонятливых), я привожу эту матрицу.

$$M_r = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$p = \begin{bmatrix} x & y & 1 \end{bmatrix},\ p' = pM_r = \begin{bmatrix} x\cos\theta - y\sin\theta & x\sin\theta + y\cos\theta & 1 \end{bmatrix}.$$

Прежде чем переходить к рассмотрению многоугольников, обратимся к упоминавшемуся выше вопросу: можно ли вместо матрицы 3×3 использовать для умножения матрицу размерностью 3×2? При этом необходимо просто добавлять к результату умножения третий член вектора, который всегда равен 1. В принципе такой вариант возможен, но я предпочитаю использовать однородные матрицы и координаты. Кстати, этот последний член может не быть равен 1, но тогда для получения конечного результата придется поделить все члены полученного вектора на его величину. Значение 1 позволяет избежать излишних операций деления.

В принципе вы можете работать с парой новых структур данных и сохранять все точки в виде вектора 1×2, а матрицы преобразований — как матрицы 3×2, добавляя при необходимости фиктивный третий член вектора.

```
// Матрица преобразования
typedef struct MATRIX3X2_TYP
{
  float M[3][2];  // Массив для хранения данных
} MATRIX3X2, *MATRIX3X2_PTR

// структура данных для двухмерной точки
typedef  struct MATRIX1X2_TYP
{
  float M[2];     // Массив для хранения данных
} MATRIX1X2, * MATRIX1X2_PTR

int Mat_Mul_1x2_3x2(MATRIX1X2_PTR ma,
            MATRIX3X2_PTR mb,
            MATRIX1X2_PTR mprod )
{
  // Эта функция умножает матрицу 1x2 на матрицу 3x2
```

```
  // и сохраняет результат, используя фиктивный элемент
  // в качестве третьего элемента матрицы 1x2, чтобы не
  // нарушать правила умножения матриц, т.е. получить
  // умножение матриц 1x3 X 3x2.

  for(int col=0; col<2; col++)
  {
    // Вычисляем скалярное произведение
    // строки ma и столбца mb
    float sum = 0;  // Результат
    for(int index = 0; index < 2; index++)
    {
      // Добавляем следующее произведение
      sum += (ma->M[index]*mb->M[index][col]);
    } // for index

    // добавляем последний элемент, умноженный на 1
    sum += = mb[index][col];

    // помещаем результирующий элемент col в массив
    mprod->M[col] = sum;

  } // for col

  return (1);
} // Mat_Mul_1x2_3x2
```

Демонстрационная программа с использованием матриц называется DEMO8_6.CPP. Я создал многоугольник, который похож на некоторый летательный аппарат, сделанный из проволоки. Вы можете его масштабировать, поворачивать и переносить. Копия экрана программы показана на рис. 8.26.

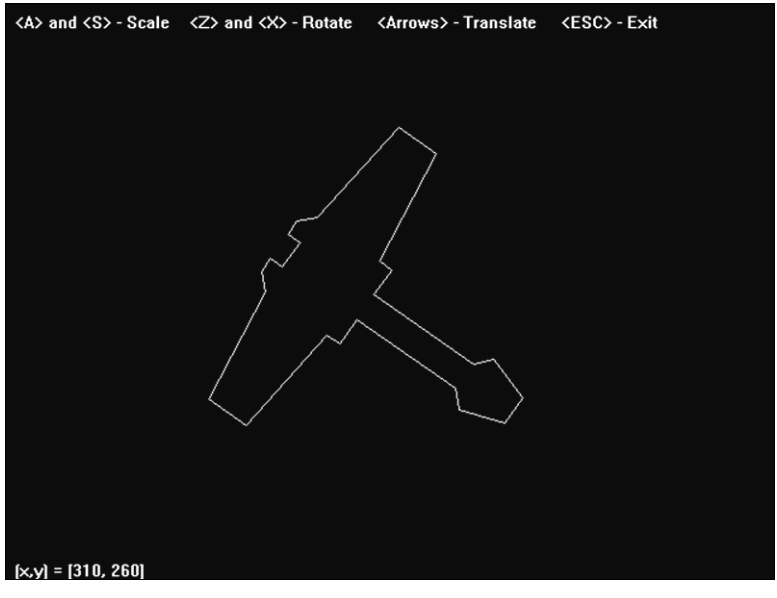


Рис. 8.26. Демонстрационная программа DEMO8_6.EXE в действии

НА ЗАМЕТКУ

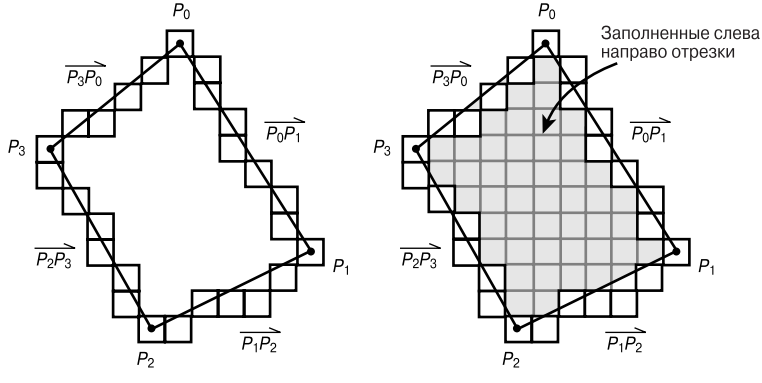
Перемножаемые матрицы всегда должны иметь размерности вида $m \times r$ и $r \times n$, т.е., другими словами, должны быть равны внутренние размеры матриц. Очевидно, что умножение матриц 1×2 и 3×2 не сработает, поскольку 2 не равно 3. Однако в приведенном выше коде в матрицу 1×2, чтобы не нарушать правила умножения матриц, добавлен фиктивный элемент, в результате чего она становится матрицей 1×3.

Управляющие клавиши этой демонстрационной программы:

<Esc>	Выход из демонстрационной программы
<A>	Увеличение размера объекта на 10%
<S>	Уменьшение размера объекта на 10%
<Z>	Вращение против часовой стрелки
<X>	Вращение по часовой стрелке
Клавиши перемещения курсора	Перенос вдоль осей X и Y

Сплошные заполненные многоугольники

Давайте отвлечемся от математики и обратимся к более реальным вещам. Одной из центральных проблем трехмерных виртуальных машин, а также многих двухмерных машин является рисование сплошных, или заполненных, многоугольников, пример которого показан на рис. 8.27. Решением этой проблемы мы сейчас и займемся.



а) Векторный или каркасный многоугольник б) Заполненный многоугольник

Рис. 8.27. Заполнение многоугольника

Нарисовать заполненные многоугольники можно различными способами. Но поскольку нашей целью является создание двух- или трехмерных игр, нам, скорее всего, понадобится рисовать многоугольник, заполненный одним цветом или текстурой. Такие многоугольники изображены на рис. 8.28. Заполнение текстурой — более сложная процедура, так что сейчас просто выясним, каким образом можно нарисовать сплошной многоугольник любого цвета.

Прежде чем приступить к решению проблемы, вы должны четко определить ее рамки. Давайте ограничимся только выпуклыми многоугольниками. Затем необходимо решить, насколько сложным будет многоугольник, т.е. сколько сторон он будет иметь — три, четыре или произвольное число? Задача имеет свое решение в каждом конкретном случае, поэтому, если

многоугольник имеет более трех сторон, должен использоваться некоторый другой алгоритм (например, четырехугольники могут быть разделены на два треугольника).

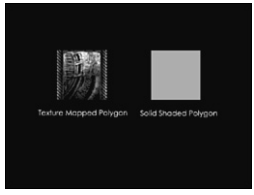


Рис. 8.28. Сплошной заштрихованный многоугольник и многоугольник, заполненный текстурой

Здесь я намерен показать, как заполняются треугольники и произвольные многоугольники.

Типы треугольников и четырехугольников

Давайте вначале рассмотрим произвольный четырехугольник, показанный на рис. 8.29. Такой четырехугольник может быть разбит на два треугольника: T_a и T_b; при этом задача вывода заполненного четырехугольника сводится к задаче вывода двух заполненных треугольников. Итак, теперь нам нужно сосредоточить все внимание на выводе одного треугольника, а затем использовать этот алгоритм для вывода треугольников и четырехугольников. Давайте рассмотрим рис. 8.30 и примемся за дело.

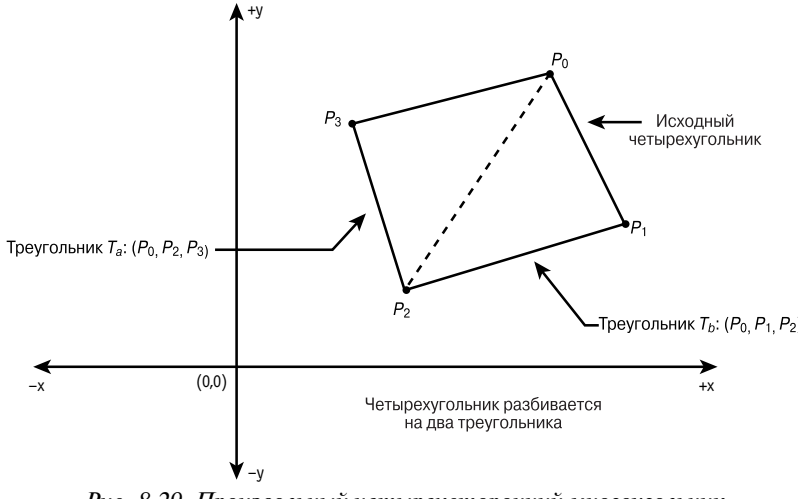


Рис. 8.29. Произвольный четырехсторонний многоугольник

Все треугольники можно разделить на четыре типа.

- **Плоский верх**. Это треугольник с горизонтальной верхней стороной, иными словами, две его верхние вершины имеют одинаковые значения координаты Y.
- **Плоский низ**. Это треугольник с горизонтальной нижней стороной, иными словами, две его нижние вершины имеют одинаковые значения координаты Y.
- **Большая правая сторона**. Это треугольник, все три вершины которого имеют различные значения координаты Y, но самая длинная сторона треугольника имеет наклон вправо.
- **Большая левая сторона**. Это треугольник, все три вершины которого имеют различные значения координаты Y, но самая длинная сторона треугольника имеет наклон влево.

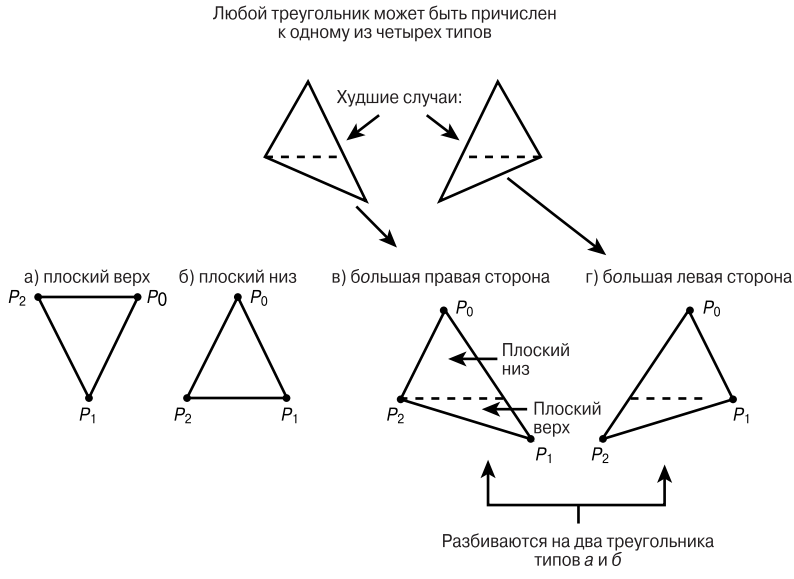


Рис. 8.30. Общие типы треугольников

Треугольники первых двух типов являются самыми простыми для растеризации, но и треугольники последних двух типов станут такими же простыми в том случае, если вы вначале разделите их на два треугольника: с плоским низом и верхом. Это можно делать или не делать, но я обычно поступаю именно так. Если не выполнить такое разделение, придется учитывать изменение наклона при переходе от одной стороны треугольника к другой. В общем, все станет намного понятнее, если мы обратимся к конкретным примерам.

Черчение треугольников и четырехугольников

Черчение треугольника во многом похоже на черчение линии в том смысле, что вы должны нанести пиксели вычерчиваемых сторон, а затем соединить их линиями. На рис. 8.31 этот процесс показан для треугольника с плоским низом. Очевидно, что если мы знаем наклон каждой стороны, то можем просто опускаться вниз, переходя от одной строки развертки к другой, и корректировать значения координат X конечных точек (т.е. x_s и x_e), исходя из величины наклона, а затем чертить соединительную линию между этими точками.

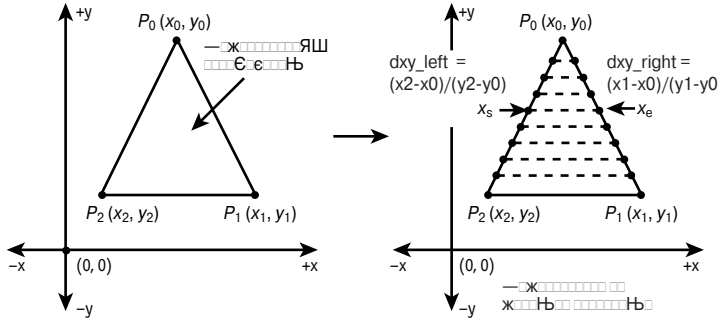


Рис. 8.31. Растеризация треугольника

Нам не нужно применять алгоритм Брезенхама, поскольку нас не интересует вычерчивание линии. Нам просто нужно знать центры пикселей, находящиеся (через целочисленные интервалы) в тех точках, где строка развертки пересекает стороны треугольника. Ниже приводится описание алгоритма, предназначенного для заполнения треугольника, имеющего плоский низ.

1. Во-первых, вычислим отношение d_x/d_y для левой и правой сторон треугольника. По сути, это значение представляет собой величину, обратную наклону. Мы хотим знать изменения значения x для каждого значения координаты y (которое и есть отношением d_x/d_y). Назовем эти величины dxy_left и dxy_right для левой и правой сторон соответственно.
2. Начиная с верхней вершины, имеющей координаты (x_0, y_0), зададим начальные значения $x_s = x_e = x_0$ и $y = y_0$.
3. Добавим dxy_left к x_s и dxy_right к x_e, отслеживая таким образом отображаемые конечные точки горизонтальных линий.
4. Чертим линию, начинающуюся в точке (x_s, y) и заканчивающуюся в точке (x_e, y).
5. Повторяем шаги 3–4 до тех пор, пока треугольник не будет растеризован по всей высоте — начиная от вершины и заканчивая горизонтальным низом.

Конечно, задание начальных и граничных условий требует особой осторожности, но это все, что нужно в данном случае. Теперь, прежде чем переходить к рассмотрению следующих вопросов, уделим немного внимания оптимизации.

На первый взгляд может показаться, что для вычерчивания сторон можно использовать операции с числами с плавающей точкой. Проблема заключается не в том, что математика с плавающей точкой более медленная, чем целочисленная математика, а в том, что в какой-то момент вам придется преобразовывать числа с плавающей точкой в целые.

Если вы позволите сделать это компилятору, ему потребуется на все вычисления примерно 60 тактов. Если вы самостоятельно напишете соответствующий код с использованием команд сопроцессора, то сможете потратить на это примерно 10–20 тактов (не забывайте, что вам не только нужно преобразовать числа с плавающей точкой в целые числа, но и сохранить их). Поэтому, несмотря на то что в демонстрационных целях используется версия с плавающей точкой, окончательная программа должна использовать математику с фиксированной точкой (о которой я расскажу вам немного позже).

Рассмотрим реализацию растеризации треугольника с плоским низом, использующую числа с плавающей точкой. Вот этот алгоритм (здесь применяются те же обозначения, что и на рис. 8.31):

```
// Вычисляем приращения
float dxy_left  = (x2-x0)/(y2-y0);
float dxy_right = (x1-x0)/(y1-y0);

// Начальная и конечная точки вычерчиваемой
// линии треугольника
float  xs = x0;
float  xe = x0;

// Чертим все строки развертки
for(int y=y0; y <= y1; y++)
{
  // Чертим линию от xs до xe для
  // заданного значения y цветом c
```

```
  Draw_Line((int)xs, (int)xe, y, c);

  // Опускаемся вниз на одну строку развертки
  xs += dxy_left;
  xe += dxy_right;
} // for
```

А теперь поговорим о работе этого алгоритма и его недостатках. Первый состоит в том, что этот алгоритм отбрасывает дробную часть конечных точек каждой строки развертки. Это скорее плохо, чем хорошо, так как приводит к потере информации. Лучше было бы округлять значение каждой конечной точки, добавляя к нему 0.5 перед преобразованием его в целое число. Другая проблема связана с начальными условиями. В ходе первой итерации алгоритм вычерчивает линию шириной в один пиксель. Хотя такая схема работает нормально, здесь, безусловно, есть возможность для оптимизации.

Осталось посмотреть, сможете ли вы теперь написать алгоритм для треугольника с плоским верхом, используя всю полученную информацию. Вам нужно обозначить вершины так же, как на рис. 8.31, а затем немного изменить начальные условия алгоритма с тем, чтобы корректно вычислять значения, интерполирующие правую и левую стороны. В результате будет получен следующий код:

```
// Вычисляем приращения
float dxy_left  = (x2-x0)/(y2-y0)
float dxy_right = (x2-x1)/(y2-y1)

// Начальная и конечная точки вычерчиваемой
// линии треугольника
float  xs = x0;
float  xe = x1;

// Чертим все строки развертки
for(int y = y0; y <= y2; y++)
{
  // Чертим линию от xs до xe для
  // заданного значения y цветом c
  Draw_Line((int)(xs+0.5),(int)(xe+0.5),y,c);

  // Опускаемся вниз на одну строку развертки
  xs += dxy_left;
  xe += dxy_right;

} // for
```

Итак, пройдена половина пути. Теперь вы можете вычерчивать треугольник, имеющий плоский верх или низ, и знаете, что треугольник, который не имеет плоского верха или низа, может быть разбит на такие треугольники. Рассмотрим эту проблему и способы ее решения.

На рис. 8.32 показан треугольник с б*о*льшей правой стороной. Понятно, что если вы сможете выполнить растеризацию треугольника с б*о*льшей правой стороной, то аналогичная операция для треугольника с б*о*льшей левой стороной не вызовет никаких затруднений. Вначале следует заметить, что в данном случае можно начать алгоритм так же, как и для треугольника с плоским низом, т.е. запуская интерполяцию сторон от верхней вершины. Проблема возникает тогда, когда, завершив интерполяцию левой стороны, мы достигаем второй вершины. Именно здесь нужно внести в работу алгоритма некоторые

изменения. В сущности, требуется еще раз вычислить интерполирующие величины (для следующей стороны) и продолжить растеризацию.

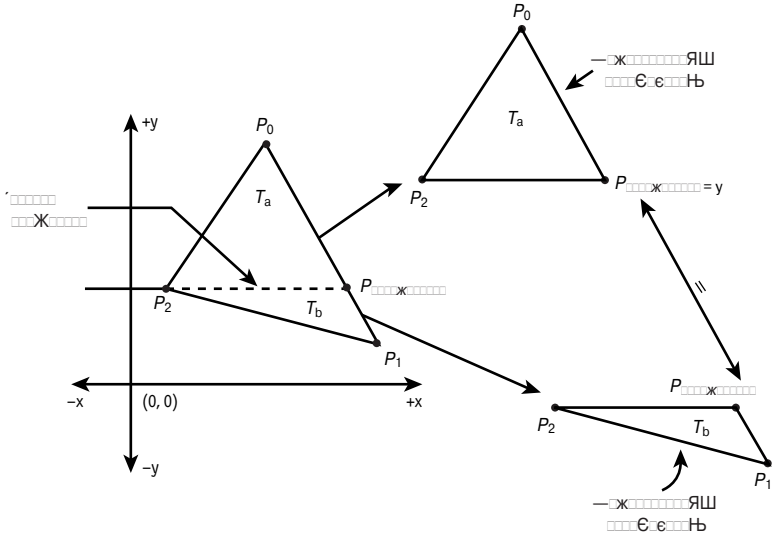


Рис. 8.32. Растеризация треугольника с большей правой стороной

Существует несколько способов решения этой проблемы. Во вложенном цикле можно начертить первую часть треугольника до точки изменения наклона, заново рассчитать величину наклона следующей стороны и продолжить процесс. Можно разбить треугольник на два (с плоским низом и верхом), а затем вызвать соответствующие функции вывода треугольников.

Воспользуемся именно этим, последним, методом. Если впоследствии обнаружится, что он не адекватен при работе в трехмерном мире, применим первый метод.

Подробности процесса разделения треугольника

Прежде чем представить вашему вниманию код, который выводит замкнутый закрашенный треугольник в 8-битовом режиме, хотелось бы проанализировать некоторые подробности, касающиеся правильного написания алгоритма.

Разделение треугольника на два других треугольника — с плоскими низом и верхом — нельзя назвать простым делом. По сути, мы продвигаемся по всей высоте короткой стороны до первой точки, где имеет место изменение наклона, а затем используем эту высоту для нахождения точки разделения треугольников на длинной стороне.

Результат аналогичен выполненному вручную перемещению вниз по длинной стороне треугольника, когда осуществляется переход с одной строки развертки на другую. Затем, когда у нас есть соответствующая точка разделения треугольника, мы просто вызываем функции растеризации треугольников с плоским верхом и низом, и треугольник готов! На рис. 8.32 показаны детали алгоритма разделения треугольника.

Помимо разделения, возникает еще одна небольшая проблема — повторное рисование одной и той же строки. Когда вы передаете общие вершины треугольника с плоской вершиной и плоским основанием, одна строка развертки, которая является для них общей, будет вычерчена дважды. Это нельзя считать очень большой проблемой, но об этом следует подумать. Чтобы избежать повторного вычерчивания общей линии треугольников, вы можете опуститься вниз на одну строку развертки в треугольнике с плоским низом.

Это почти все, если не считать такой мелочи, как отсечение... Надеюсь, вы помните о двух способах отсечения: в *пространстве объектов* и в *пространстве изображений*. Пространство объектов — это замечательно, но если вам нужно отсечь треугольник прямоугольником экрана, в худшем случае вам придется добавлять четыре дополнительные вершины. Эта ситуация показана на рис. 8.33.

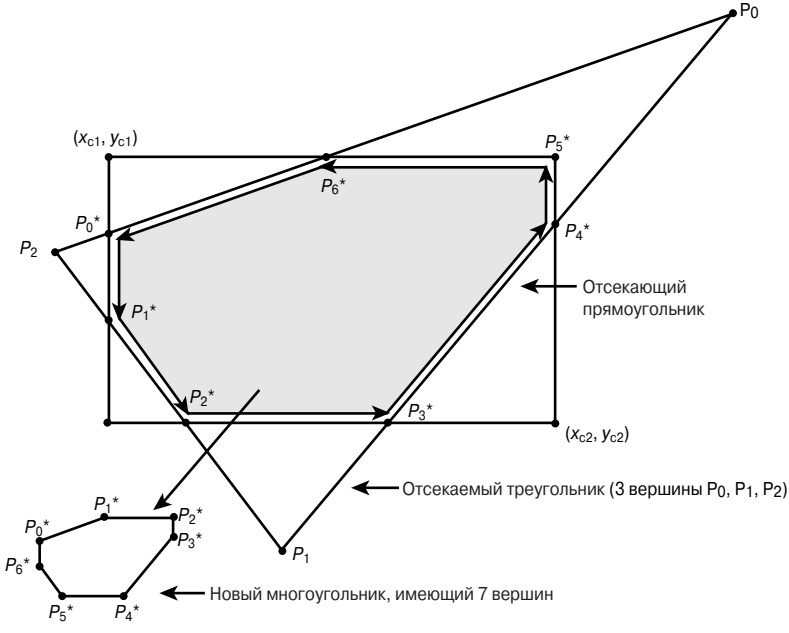


Рис. 8.33. Наихудший случай отсечения треугольника

Лучше пойти по более легкому пути и выполнять отсечение в пространстве изображений по мере рисования многоугольника, но при этом будут отсекаться не отдельные пиксели, а строка развертки. Чтобы определить, нужно ли вообще делать отсечение, можно провести некоторые стандартные тесты, и если отсечение делать не нужно, то будет выполняться та часть программы, где вывод осуществляется без отсечения и потому работает быстрее.

Если же остановиться на стандартном варианте отсечения, то придется заниматься всеми случаями вырождения треугольника, например в одиночную точку либо прямую горизонтальную или вертикальную линию. Код должен будет содержать проверку всех этих случаев, а кроме того, ему придется заниматься сортировкой передаваемых функции координат вершин треугольника, упорядочивая вершины сверху вниз и слева направо. С учетом всего этого рассмотрим три функции вычерчивания треугольников в 8-битовом режиме.

Функция черчения треугольника с плоским верхом:

```
void Draw_Top_Tri(int x1, int y1,
                  int x2, int y2,
                  int x3, int y3,
                  int color,
                  UCHAR *dest_buffer, int mempitch )
{
  // Вывод треугольника с плоским верхом
  float dx_right, // Отношение dx/dy для правой линии
        dx_left,  // Отношение dx/dy для левой линии
```

```
    xs, xe,      // Начальная и конечная точки линии
    height;   // Высота треугольника

int temp_x,   // Временные переменные при сортировке
   temp_y,
   right,       // Используются при отсечении
   left;

// Адрес очередной строки развертки
UCHAR *dest_addr = NULL;

// Упорядочиваем x1 и x2
if (x2 < x1)
{
   temp_x = x2;
   x2 = x1;
   x1 = temp_x;
} // if

// Вычисляем приращения
height = y3 - y1;

dx_left  = (x3-x1)/height;
dx_right = (x3-x2)/height;

// Задаем начальные точки
xs = (float)x1;
xe = (float)x2+(float)0.5;

// Выполняем отсечение по оси y
if (y1 < min_clip_y)
{
   // Вычисляем новые xs и ys
   xs = xs+dx_left*(float)(-y1+min_clip_y);
   xe = xe+dx_right*(float)(-y1+min_clip_y);

   // Сбрасываем значение y1
   y1 = min_clip_y;

} // if

if (y3 > max_clip_y)
   y3 = max_clip_y;

// Вычисляем начальный адрес в видеопамяти
dest_addr = dest_buffer+y1*mempitch;

// Проверяем, нужно ли отсечение по оси x
if (x1 >= min_clip_x && x1 <= max_clip_x &&
   x2 >= min_clip_x && x2 <= max_clip_x &&
   x3 >= min_clip_x && x3 <= max_clip_x )
{
   // Чертим треугольник
   for(temp_y = y1; temp_y<= y3;
       temp_y++, dest_addr += mempitch)
```

```
    {
      memset((UCHAR* )dest_addr+(unsigned int)xs,
           color, (unsigned  int)(xe-xs+1));

      // Скорректируем начальную и конечную точки
      xs += dx_left;
      xe += dx_right;
    } конец for
  } // if
  else
  {
    // Отсечение по оси x (медленная версия)

    // Чертим треугольник
    for (temp_y = y1; temp_y <= y3;
        temp_y++, dest_addr += mempitch)
    {
      // Делаем отсечение по x
      left  = (int)xs;
      right = (int)xe;

      // Корректируем начальную и конечную точки
      xs += dx_left;
      xe += dx_right;

      // Отсекаем линию
      if (left < min_clip_x)
      {
        left = min_clip_x;
        if (right < min_clip_x)
          continue;
      }

      if (right > max_clip_x)
      {
        right = max_clip_x;
        if (left > max_clip_x)
          continue;
      }

      memset((UCHAR* )dest_addr+(unsigned int)left,
          color, (unsigned int)(right-left+1));
    } // for
  } // else
} // Draw_Top_Tri
```

Функция черчения треугольника с плоским низом:

```
void Draw_Bottom_Tri (int x1, int y1,
                int x2, int y2,
                int x3, int y3,
                int color,
                UCHAR* dest_buffer, int mempitch)
{
  // Функция выводит треугольник с плоским низом
```

```
float dx_right,   // Отношение dx/dy для правой линии
      dx_left,    // Отношение dx/dy для левой линии
      xs, xe,     // Начальная и конечная точки сторон
      height;     // Высота треугольника

int temp_x,       // Временные переменные сортировки
    temp_y,
    right,        // Используются при отсечении
    left;

// Адрес очередной строки развертки
UCHAR* dest_addr;

// Упорядочиваем x2 и x3
if (x3 < x2)
{
   temp_x = x2;
   x2 = x3;
   x3 = temp_x;
} // if

// Вычисляем приращения
height = y3 - y1;

dx_left  = (x2-x1)/height;
dx_right = (x3-x1)/height;

// Задаем начальные точки
xs = (float)x1;
xe = (float)x1; //  +(float)0.5;

// Выполняем отсечение по оси y
if (y1 < min_clip_y)
{
   // Вычисляем новые xs и ys
   xs = xs+dx_left*(float)(-y1+min_clip_y);
   xe = xe+dx_right*(float)(-y1+min_clip_y);

   // Сбрасываем значение y1
   y1 = min_clip_y;

} // if

if (y3 > max_clip_y)
   y3 = max_clip_y;

// Вычисляем начальный адрес в видеопамяти
dest_addr = dest_buffer+y1*mempitch;

// Проверяем, нужно ли отсечение по оси x
if (x1 >= min_clip_x && x1 <= max_clip_x &&
   x2 >= min_clip_x && x2 <= max_clip_x &&
   x3 >= min_clip_x && x3 <= max_clip_x)
{
   // Чертим треугольник
   for (temp_y = y1; temp_y <= y3;
```

```
        temp_y++, dest_addr += mempitch)
    {
      memset((UCHAR*)dest_addr+(unsigned int)xs,
           color, (unsigned  int)(xe-xs+1));

      // Корректируем начальную и конечную точки
      xs += dx_left;
      xe += dx_right;
    } // for
  }  // if
  else
  {
    // Отсечение по оси x (медленная версия)

    // Чертим треугольник
    for (temp_y = y1; temp_y <= y3;
        temp_y++, dest_addr += mempitch)
    {
      // Отсечение по x
      left  = (int)xs;
      right = (int)xe;

      // Корректируем начальную и конечную точки
      xs += dx_left;
      xe += dx_right;

      // Отсекаем линию
      if (left < min_clip_x)
      {
        left = min_clip_x;
        if (right < min_clip_x)
          continue;
      }

      if (right > max_clip_x)
      {
        right = max_clip_x;
        if (left > max_clip_x)
          continue;
      }

      memset((UCHAR*)dest_addr+(unsigned  int)left,
          color, (unsigned  int) (right - left+1));
    } // for
  }  // else
}  // Draw_Bottom_Tri
```

И наконец, последняя функция чертит произвольный треугольник, разделяя его при необходимости на треугольники с плоским верхом и низом:

```
void Draw_Triangle_2D(int x1, int y1,
                      int x2, int y2,
                      int x3, int y3,
                      int color,
                      UCHAR* dest_buffer, int mempitch)
```

```
{
  // Функция чертит треугольник путем разбиения на пары
  // треугольников с плоским верхом и низом

  int temp_x, // Временные переменные сортировки
    temp_y,
    new_x;

  // Проверка линий на горизонтальность и вертикальность
  if ((x1 == x2 && x2 == x3) || (y1 == y2 && y2 == y3))
    return;

  // Сортируем p1 и p2 в порядке возрастания значения y
  if (y2 < y1)
  {
    temp_x = x2;
    temp_y = y2;
    x2    = x1;
    y2    = y1;
    x1    = temp_x;
    y1    = temp_y;
  } // if

  // Теперь проверим порядок точек p1 и p3
  if (y3 < y1)
  {
    temp_x = x3;
    temp_y = y3;
    x3    = x1;
    y3    = y1;
    x1    = temp_x;
    y1    = temp_y;
  } // if

  // Наконец, проверим порядок p3 и p2
  if (y3 < y2)
  {
    temp_x = x3;
    temp_y = y3;
    x3    = x2;
    y3    = y2;
    x2    = temp_x;
    y2    = temp_y;
  } // if

  // Стандартные проверки, выполняемые при отсечении
  if (y3<min_clip_y || y1>max_clip_y ||
    (x1<min_clip_x && x2<min_clip_x && x3<min_clip_x)||
    (x1<max_clip_x && x2<max_clip_x && x3<max_clip_x))
    return;

  // Проверяем, является ли верх треугольника плоским
  if (y1 == y2)
  {
```

```
    Draw_Top_Tri(x1, y1, x2, y2, x3, y3, color,
          dest_buffer, mempitch);
  } // if
  else
    if (y2 == y3)
    {
      Draw_Bottom_Tri(x1, y1, x2, y2, x3, y3, color,
            dest_buffer, mempitch);
    } // if
    else
    {
      // Произвольный разделяемый треугольник; линия
      // разделения пересекает длинную сторону
      new_x = x1+(int)(0.5+(float)(y2-y1)*
          (float)(x3-x1)/(float)(y3-y1));

      // Чертим треугольники, полученные в результате
      // разбиения
      Draw_Bottom_Tri(x1,y1,new_x,y2,x2,y2,color,
            dest_buffer,mempitch);
      Draw_Top_Tri(x2,y2,new_x,y2,x3,y3,color,
          dest_buffer,mempitch);
    } // else
} // Draw_Triangle_2D
```

Для работы вы должны вызывать последнюю функцию, поскольку в ней содержатся обращения к другим функциям поддержки. Приведу пример вызова функции, которая рисует треугольник с координатами (100,100), (200, 150), (40, 200) цветом 30:

```
Draw_Triangle_2D(100, 100, 200, 150, 40, 200, 30,
        back_buffer, back_pitch);
```

Вообще говоря, вы должны передавать координаты в порядке, соответствующем направлению обхода вершин треугольника против часовой стрелки. В данный момент это никакого значения не имеет, но в трехмерном представлении эта деталь приобретает важное значение, так как ряд алгоритмов, имеющих дело с трехмерным представлением, осуществляют проверку порядка обхода вершин для определения лицевой и невидимой поверхностей многоугольника.

СЕКРЕТ

Помимо упомянутых функций черчения многоугольников, я написал их версии для чисел с фиксированной точкой, которые работают немного быстрее. Эти функции содержатся в библиотечном файле `T3DLIB1.CPP`. Их имена заканчиваются символами `FP`, но работают они точно так же, как и функции с плавающей точкой. Как правило, вам нужно вызывать только одну такую функцию — `Draw_TriangleFP_2D()`. Она создает такое же изображение, как и функция `Draw_Triangle_2D()`, но работает существенно быстрее. Если вас интересует математика с фиксированной точкой, обратитесь к главе 11, "Алгоритмы, структуры данных, управление памятью и многопоточность".

Демонстрационная программа `DEMO8_7.CPP` на прилагаемом компакт-диске представляет собой пример работы функции вычерчивания многоугольника. Она в 8-битовом режиме вычерчивает случайные отсеченные треугольники. Глобальная отсекающая область задается в описанных далее переменных.

```
int min_clip_x = 0; // Отсекающий прямоугольник
  max_clip_x  = (SCREEN_WIDTH-1),
```

```
  min_clip_y = 0;
  max_clip_y = (SCREEN_HEIGH-1);
```

А теперь рассмотрим более сложные способы растеризации, которые применяются для многоугольников, имеющих более трех вершин.

Общий случай растеризации четырехугольника

Как вы могли заметить, растеризация простого треугольника — не самая легкая работа в этом мире. Поэтому вполне можно предположить, что растеризация многоугольников, имеющих более трех вершин, является еще более сложной.

Однако растеризация четырехугольника не будет очень трудной, если вы разделите его на два треугольника. В качестве примера вернемся к рис. 8.29, где изображен четырехугольник, разделенный на два треугольника. В сущности, вы можете использовать этот простой алгоритм для разделения любого четырехугольника на два треугольника.

Пусть вершины многоугольника, заданные в некотором порядке (например, по часовой стрелке), обозначены цифрами 0, 1, 2, 3. Тогда

треугольник 1 содержит вершины 0, 1, 3;

треугольник 2 содержит вершины 1, 2, 3.

Я написал для вас функцию, выполняющую такое разделение, и назвал ее Draw_QuadFP_2D() (версия этой функции для чисел с плавающей точкой отсутствует). Вот как выглядит исходный текст данной функции:

```
inline void Draw_QuadFP_2D(int x0, int y0,
                           int x1, int y1,
                           int x2, int y2,
                           int x3, int y3,
                           int color,
                           UCHAR* dest_buffer,
                           int mempitch)
{
  // Эта функция чертит плоский четырехугольник

  // Вызываем два раза функцию черчения треугольника
  // и позволяем ей сделать всю необходимую работу
  Draw_TriangleFP_2D(x0, y0, x1, y1, x3, y3, color,
                     dest_buffer, mempitch);
  Draw_TriangleFP_2D(x1, y1, x2, y2, x3, y3, color,
                     dest_buffer, mempitch);
} // Draw_QuadFP_2D
```

Эта функция идентична функции вывода треугольника, с тем исключением, что она принимает в качестве параметров на одну вершину больше. Демонстрационная программа на прилагаемом компакт-диске, показывающая работу этой функции, носит название DEMO8_8.CPP. В ходе ее выполнения на экране рисуется ряд произвольных четырехугольников.

НА ЗАМЕТКУ

Здесь я допустил некоторую небрежность по отношению к параметрам. Вероятно, было бы намного лучше определить структуру данных для многоугольника, а затем передать в функцию адрес такой структуры. Я оставляю код как есть, но вы должны иметь это в виду, поскольку вам придется использовать такой способ задания многоугольника при переходе к трехмерному представлению.

Триангуляция многоугольников

Итак, вы умеете рисовать треугольник и четырехугольник. Но как нарисовать многоугольник, имеющий более четырех вершин? Вы могли бы триангулировать многоугольник так, как показано на рис. 8.34. Хотя этот метод вполне хорош и во многих графических подсистемах (особенно в аппаратных средствах) применяется именно такой способ, подобный подход достаточно сложен для решения задачи в общем виде.

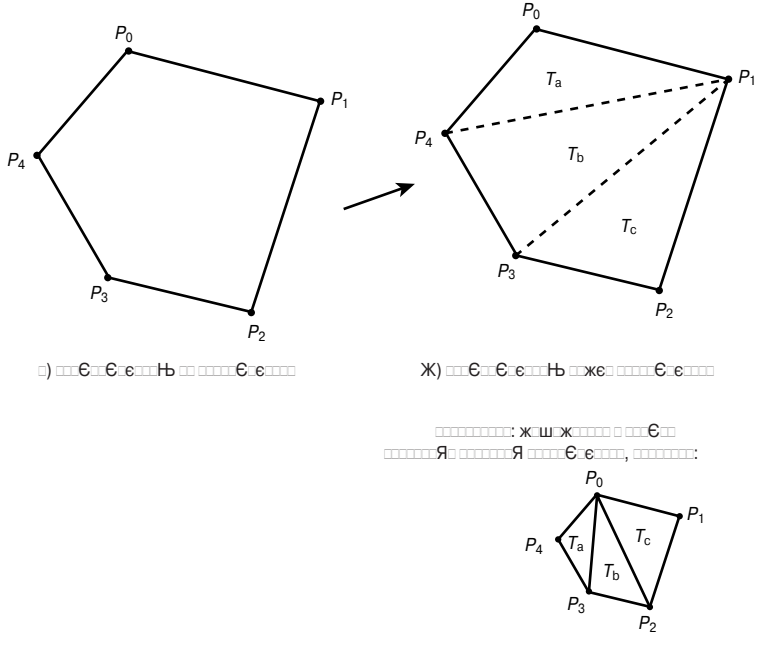


Рис. 8.34. Триангуляция большого многостороннего многоугольника

Если многоугольник всегда выпуклый, задача в значительной степени упрощается. Для ее решения придумано множество алгоритмов, но я обычно применяю один алгоритм, который является рекурсивным по своей природе и весьма прост. На рис. 8.35 показана последовательность действий при триангуляции выпуклого пятиугольника.

По поводу рис. 8.35 замечу, что существует несколько возможных вариантов триангуляции. Таким образом, здесь для оптимизации процесса триангуляции можно использовать некоторую эвристику и/или оценочные функции некоторого рода. Неплохой, например, может оказаться идея разделения на треугольники примерно одинаковой площади или выделения в первую очередь самых больших треугольников.

В любом случае здесь есть над чем подумать. А сейчас рассмотрим некоторый обобщенный алгоритм, способный выполнить всю необходимую работу.

Пусть задан многоугольник, имеющий *n* вершин (*n* может быть четным или нечетным), порядок которых соответствует направлению по или против часовой стрелки, и вам нужно разбить этот многоугольник на треугольники.

1. Если число вершин, которые осталось обработать, больше трех, то продолжаем и переходим к шагу 2; в противном случае выходим из программы.
2. Рассмотрим первые три вершины как принадлежащие некоторому треугольнику.
3. Выделяем новый треугольник и рекурсивно повторяем шаг 2 для остальных (*n*-1) вершин.

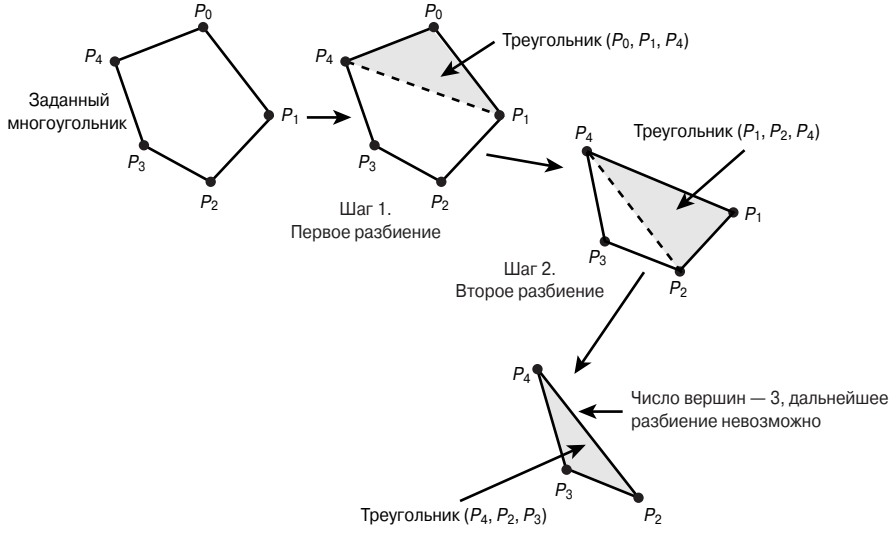


Рис. 8.35. Схема алгоритма одного из вариантов триангуляции

По сути, этот алгоритм осуществляет постепенное выделение треугольников и последующую обработку оставшихся вершин с помощью того же алгоритма. Он очень примитивен и не выполняет никакой предварительной обработки или тестирования. Разумеется, после того как вы сделали преобразование многоугольника в треугольники, можете передавать каждый из них для растеризации и отображения на экране.

Рассмотрим теперь другой подход к решению задачи растеризации произвольного выпуклого многоугольника. Если рассмотреть проблему в контексте растеризации треугольников, то растеризация *n*-стороннего выпуклого многоугольника — это просто дело техники.

Чтобы увидеть алгоритм в действии, посмотрите на рис. 8.36. Вы должны отсортировать вершины сверху вниз и слева направо с тем, чтобы получить массив вершин, упорядоченный в соответствии с направлением по часовой стрелке. Затем, начиная с самой верхней вершины, выполнить растеризацию двух сторон (правой и левой), выходящих из этой вершины. Когда одна из сторон достигает следующей вершины (на правой или левой стороне), вы должны заново рассчитать интерполирующие величины растеризации, т.е. значения `dx_left` и `dx_right`, и продолжать так до тех пор, пока растеризация многоугольника не будет завершена полностью.

Блок-схема данного алгоритма показана на рис. 8.37. Здесь также нужно позаботиться о некоторых граничных условиях, например соблюдать осторожность, чтобы не нарушить последовательности использования интерполирующих сторон во время перехода от одной вершины к другой (но это, пожалуй, все). Здесь вы также можете выполнять отсечение, используя пространство объектов или пространство изображений. Мы очень кратко остановимся на этот вопросе.

Когда вы выполняете растеризацию треугольников, вам не хочется делать отсечение в пространстве изображений, поскольку в случае отсечения всех вершин в результате может получиться шестиугольник. Я не думаю, что это очень вас обрадует, так как опять придется разбивать новый многоугольник на треугольники.

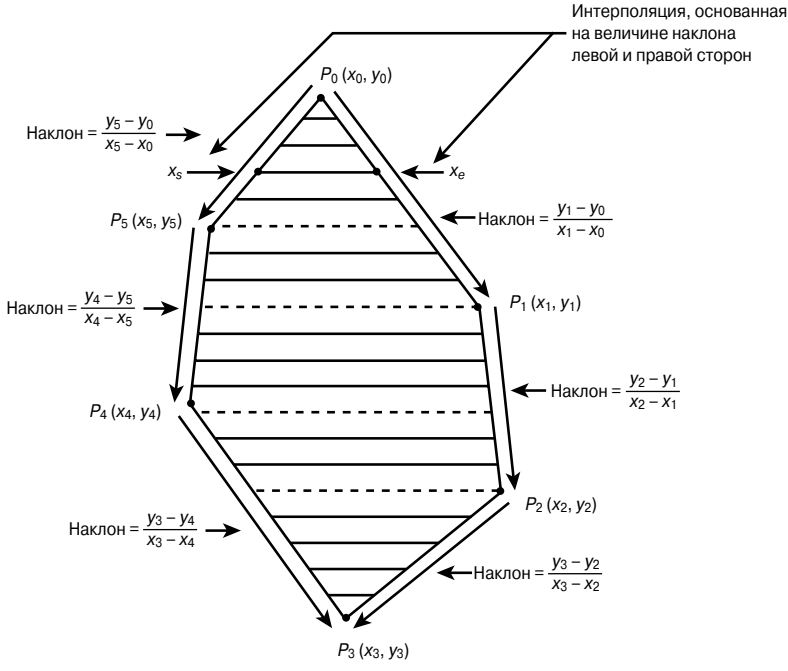


Рис. 8.36. Растеризация n*-стороннего выпуклого многоугольника без использования триангуляции*

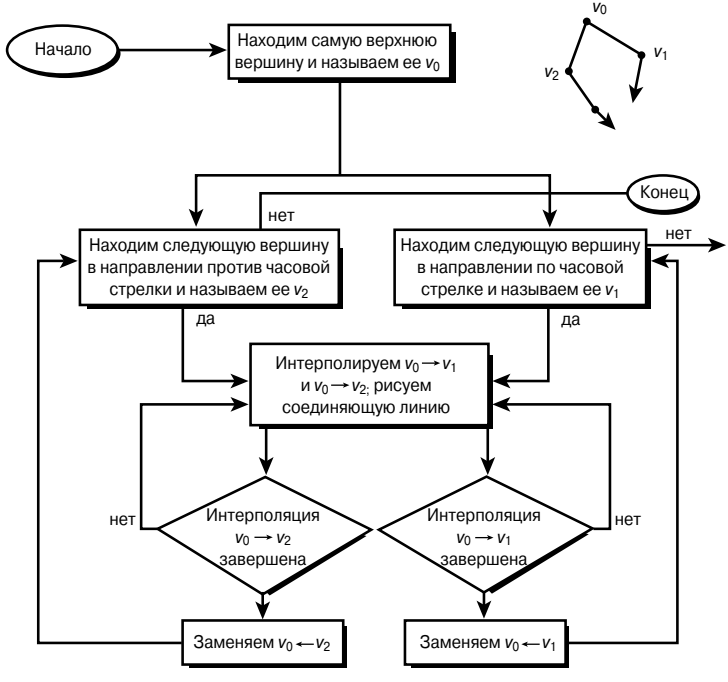


Рис. 8.37. Блок-схема алгоритма вычерчивания произвольного n*-стороннего выпуклого многоугольника*

Вы должны рассмотреть один вопрос: может ли выпуклый многоугольник в ходе операции отсечения стать невыпуклым? Безусловно. Но только в том случае, когда сама по себе отсекающая область является невыпуклой. Таким образом, отсечение выпуклого многоугольника до экранного прямоугольника в худшем случае добавит еще одну вершину к каждой из вершин, которые выходят за рамки отсекающей области.

Обычно самым лучшим способом растеризации *n*-угольника является отсечение в объектном пространстве, а затем растеризация многоугольника без отсекания линий сканирования.

В библиотеке T3DLIB имеется функция, которая принимает стандартный указатель на структуру POLYGON2D_PTR, а также адрес буфера кадров и значение шага видеопамяти, а затем растеризует переданный ей многоугольник. Разумеется, многоугольник должен быть выпуклым, и все вершины не должны выходить за рамки отсекающей области, поскольку функция не делает отсечения. Прототип этой функции имеет вид

```
void Draw_Filled_Polygon2D(POLYGON2D_PTR poly,
                UCHAR* vbuffer,
                int mempitch);
```

Чтобы нарисовать квадрат со сторонами 100×100 и центром в точке (320, 240), вы должны вызвать функцию следующим образом:

```
POLYGON2D square; // Хранит квадрат

// Задаем точки выпуклого объекта
VERTEX2DF square_vertices[4]
    = {-50, -50, 50, -50, 50, 50, -50, 50 };

// Инициализируем квадрат
object.state     = 1;
object.num_verts  = 4;
object.x0        = 320;
object.y0        = 240;
object.xv        = 0;
object.yv        = 0;
object.color     = 255;
object.vlist     = new VERTEX2DF[square.num_verts];

// Копируем вершины в структуру многоугольника
for(int index = 0; index < square.num_verts; index++)
  square.vlist[index] = square_vertices[index];

// ... вызываем функцию вывода многоугольника ...
Draw_Filled_Polygon2D(&square,(UCHAR*)ddsd.lpSurface,
            ddsd.lPitch);
```

Мне бы хотелось показать вам полный листинг этой функции, но он займет слишком много места. Но вы можете найти этот код в демонстрационной программе DEMO8_9.CPP на прилагаемом компакт-диске, которая выполняет поворот квадрата, а затем вызывает функцию его вывода. Квадрат не разбивается предварительно на два треугольника; данная функция растеризует многоугольник сразу и без отсечения.

Обнаружение столкновений при работе с многоугольниками

Мы уже рассмотрели изрядное количество информации. Теперь мне хочется сделать небольшую передышку и поговорить о некоторых вещах, связанных с играми, например об обнаружении столкновений многоугольных объектов. Я намерен ознакомить вас с тремя различными подходами к решению этой задачи. Применяя эти методы в чистом виде или объединяя их, вы будете в состоянии решить все задачи, связанные с обнаружением столкновений многоугольников.

Описанная окружность/сфера

Первый метод выявления столкновения двух многоугольников — это простое предположение, что объект обладает некоторым усредненным радиусом, и последующая проверка перекрытия сфер или окружностей с данными радиусами. Как показано на рис. 8.38, подобная проверка выполняется путем простого вычисления расстояния.

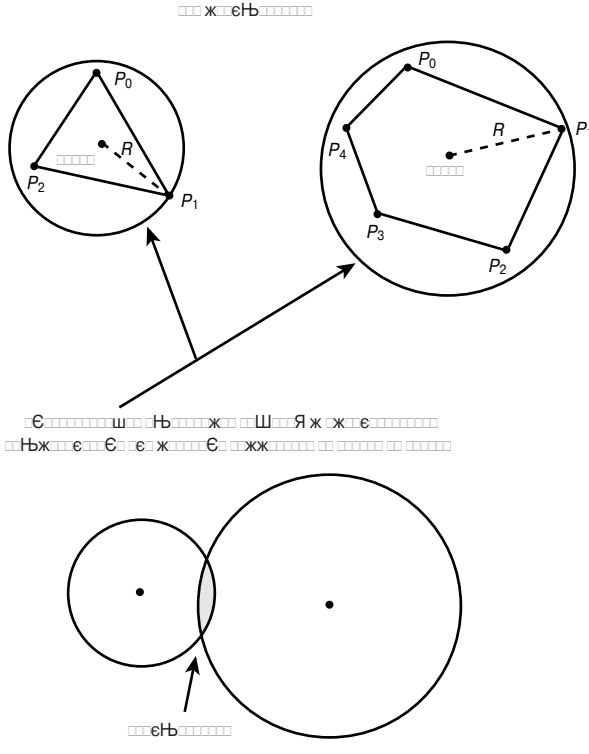


Рис. 8.38. Использование ограничивающих окружностей (в трехмерном представлении — сфер) для обнаружения столкновения

Конечно, вокруг каждого многоугольника должны быть описаны ограничивающие окружности. Недостаток этого метода в том, что в результате можно получить столкновения там, где их нет, и в то же время пропустить имеющиеся столкновения (в зависимости от способа вычисления среднего радиуса).

Чтобы реализовать этот алгоритм, вначале необходимо рассчитать значение радиуса для каждого многоугольника. Это можно сделать несколькими способами. Можно рассмотреть расстояния от центра многоугольника до каждой из вершин и затем вычислить среднее значение, или использовать самое большое значение, или применить некоторый другой эвристический подход. В любом случае подобные вычисления могут быть выполнены вне основного цикла игры, поэтому они никак не влияют на скорость работы последней. Однако сама проверка столкновения в процессе выполнения представляет собой сложную задачу.

Для вычисления расстояния между двумя точками (x_1, y_1) и (x_2, y_2) в двухмерном пространстве используйте формулу $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Для трехмерного пространства просто добавьте в выражение под знаком квадратного корня член $(z_1 - z_2)^2$.

Пусть заданы два многоугольника: poly1 с центром в точке (x_1, y_1) и poly2 с центром в точке (x_2, y_2) и радиусами r_1 и r_2 соответственно (вычисленными некоторым способом). Чтобы проверить, перекрываются ли многоугольники, можно использовать следующий псевдокод:

```
// Вычисляем расстояние между центрами многоугольников
dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));

// Проверяем расстояние
if (dist <= (r1+r2))
{
  // Столкновение!
} // if
```

Этот код будет работать вполне нормально, но здесь возникает одна проблема — он работает крайне медленно! Для вычисления квадратного корня соответствующей функции понадобится огромное количество процессорного времени, так что от него нужно как-то избавляться. В качестве разминки сначала рассмотрим более простые варианты оптимизации. Во-первых, нет никакой необходимости вычислять два раза одни и те же выражения разностей (x1-x2) и (y1-y2). Их можно вычислить один раз и затем использовать в вычислениях сохраненные результаты.

```
float dx = (x1-x2);
float dy = (y1-y2);
dist = sqrt(dx*dx+dy*dy);
```

Небольшой выигрыш есть, но функция sqrt() выполняется примерно в 70 раз дольше, чем операция умножения с плавающей точкой (т.е. функция FMUL использует примерно 1–3 цикла процессора Pentium, а функция FSQRT — около 70 циклов). Такое решение неприемлемо. Посмотрим, что можно сделать. Один из приемов состоит в том, чтобы вычислять расстояние при помощи математического метода, основанного на разложении в ряд Тэйлора–Маклорена.

Ряды Тейлора–Маклорена представляют собой математический инструмент, применяемый для аппроксимации сложных функций. Он заключается в представлении функции в виде степенного ряда. В общем случае разложение функции $f(x)$ в ряд Маклорена имеет вид:

$$f(x) = \sum_{i=0}^{\infty} f^{(i)}(0)\frac{x^i}{i!} = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + f'''(0)\frac{x^3}{3!} + \ldots,$$

где ′ обозначает *производную*, а ! — *факториал*.

С помощью этой математики, вы можете написать функцию, которая аппроксимирует расстояние между точками p_1 и p_2 в двухмерном (или трехмерном) пространстве с помощью всего лишь нескольких проверок и операций сложения. Вот алгоритмы для случаев двух- и трехмерного представления:

```
// Вычисление минимального и максимального
// значений двух выражений
#define MIN(a, b)  (((a) < (b)) ? (a) : (b))
#define MAX(a, b)  (((a) > (b)) ? (a) : (b))

#define SWAP(a,b,t) {t=a; a=b; b=t;}

int Fast_Distance_2D(int  x, int y);
{
   // Вычисление расстояния от точки 0,0 до точки
   // x,y с погрешностью 3.5%

   // Вычисляем абсолютные значения координат x,y
   x = abs(x);
   y = abs(y);

   // Вычисляем минимальное значение среди x,y
   int mn = MIN (x,y);

   // Возвращаем расстояние
   return (x+y-(mn>>1)-(mn>>2)+(mn>>4));
} // Fast_Distance_2D


float Fast_Distance_3D(float fx, float fy, float fz)
{
   // Вычисление расстояния от начала координат
   // до точки x,y,z
   int temp;   // Временная переменная
   int x,y,z;

   // Делаем все значения положительными
   x = fabs (fx) * 1024;
   y = fabs (fy) * 1024;
   z = fabs (fz) * 1024;

   // Сортируем переменные
   if (y < x) SWAP (x,y,temp);
   if (z < y) SWAP (y,z,temp);
   if (y < x) SWAP (x,y,temp);

   // Вычисляем расстояние с погрешностью до 8%
   int dist = (z+11*(y >> 5)+(x >> 2));

   return ((float) (dist >> 10));

} // Fast_Distance_3D
```

Параметрами, передаваемыми каждой функции, являются разности значений координат. Например, чтобы использовать функцию Fast_Distance_2D() в контексте предыдущего алгоритма, вы должны вызвать ее как

```
dist = Fast_Distance_2D(x1-x2, y1-y2);
```

Этот метод содержит только три операции сдвига, четыре операции сложения, несколько операций сравнения, два получения абсолютных значений и работает намного быстрее!

НА ЗАМЕТКУ

Замечу, что оба алгоритма являются приближенными, поэтому в том случае, когда необходима высокая точность, следует соблюдать осторожность. Максимальная погрешность версии алгоритма для двухмерного представления составляет 3.5%, а для трехмерного — 8%.

И еще один момент. Проницательный читатель не преминет заметить, что здесь существует еще одна оптимизация: квадратный корень можно вообще не вычислять. Я имею в виду следующее. Предположим, вам нужно определить, находится ли один объект на расстоянии 100 единиц от другого. Вы знаете, что расстояние задается формулой $d = \sqrt{x^2 + y^2}$, но, если возвести в квадрат обе части этого уравнения, мы получим $d^2 = x^2 + y^2$. В данном случае d равно 100, а $d^2 = 10\,000$. Таким образом, если при проверке выясняется, что $x^2 + y^2 < 10\,000$, это равнозначно тому, что $d < 100$ при вычислении квадратного корня. С этим способом связана только одна проблема — переполнение. Тем не менее на самом деле нет каких бы то ни было причин вычислять реальное расстояние — нас интересуют только отношения величин, так что мы вполне можем сравнивать квадраты значений.

Ограничивающий прямоугольник

Хотя математическое обоснование алгоритма ограничивающей сферы/окружности очень простое, главная проблема, возникающая в данном случае, заключается в том, что объект аппроксимируется с помощью окружности, а это иногда неадекватно. Рассмотрим, например, рис. 8.39; на нем изображен многоугольный объект, геометрия которого в целом напоминает прямоугольник. Если представить такой объект с помощью ограничивающей сферы, ошибки будут весьма значительны, поэтому лучше использовать в данном случае геометрический объект, который в большей степени похож на ограничиваемый объект. В частности, чтобы упростить обнаружение столкновения, можно использовать ограничивающий прямоугольник или квадрат.

Построение прямоугольника, ограничивающего многоугольник, выполняется точно так же, как и построение ограничивающей сферы, с тем отличием, что вместо радиуса необходимо определить стороны прямоугольника. Обычно я использую для них имена max_x, min_x, max_y, min_y, и они вычисляются относительно центра многоугольника. На рис. 8.40 определение сторон четырехугольника показано графически.

Для поиска значений переменных (max_x, min_x, max_y, min_y) можно использовать следующий простой алгоритм.

1. Присваиваем этим переменным начальные значения, равные нулю. При этом подразумевается, что центр многоугольника находится в точке (0,0).
2. Для каждой вершины (x,y) сравниваем ее координаты с величинами (max_x, min_x, max_y, min_y) и соответствующим образом обновляем последние.

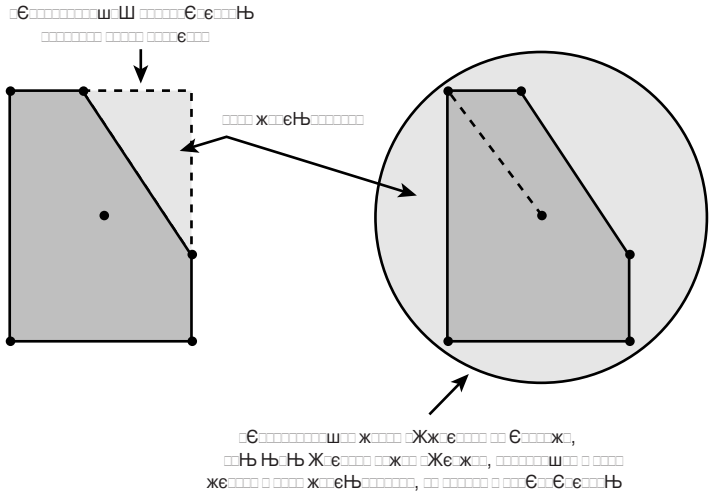


Рис. 8.39. Использование ограничивающей фигуры с наиболее подходящей геометрической формой

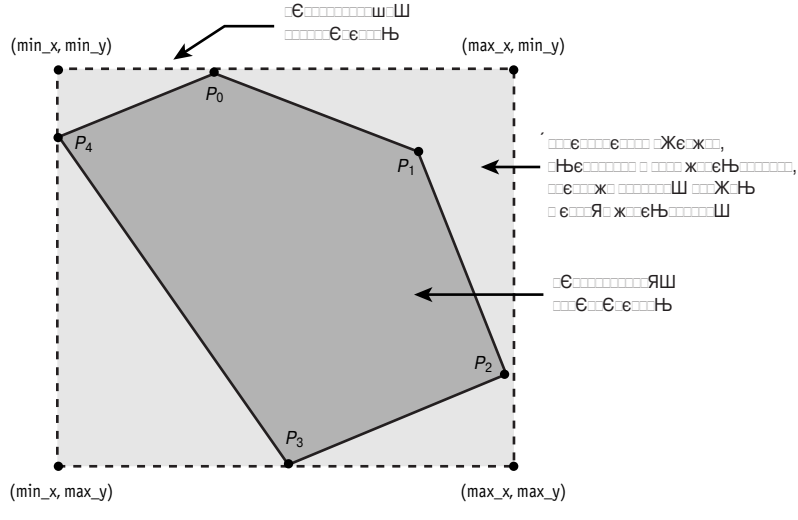


Рис. 8.40. Определение ограничивающего прямоугольника

Ниже представлен код этого алгоритма, предназначенный для работы со стандартной структурой POLYGON2D.

```
int Find_Bounding_Box_Poly2D(POLYGON2D_PTR poly,
                             float &min_x, float &max_x,
                             float &max_x, float &max_y)
{
  // Эта функция находит ограничивающий прямоугольник для
  // плоского многоугольника

  // Проверяем корректность указателя poly
```

```
  if (poly->num_verts == 0)
    return (0);

  // Инициализируем выходные данные (обратите внимание,
  // что здесь используются указатели, а также что мы
  // работаем с локальными координатами)
  max_x = max_y = min_x = min_y = 0;

  // Обрабатываем каждую из вершин
  for(int index=0; index < poly->num_verts; index++)
  {
    // обновляем переменные (поиск min/max)
    if (poly->vlist[index].x > max_x)
      max_x = poly->vlist[index].x;

    if (poly->vlist[index].x < min_x)
      min_x = poly->vlist[index].x;

    if (poly->vlist[index].y > max_y)
      max_y = poly->vlist[index].y;

    if (poly->vlist[index].y < min_y)
      min_y = poly->vlist[index].y;

  } // for
  // Возвращаем код успешного завершения функции
  return(1);
} // Find_Bounding_Box_Poly2D
```

НА ЗАМЕТКУ Обратите внимание, что эта функция принимает параметры, передаваемые по ссылке, с использованием для этого оператора &. Такие параметры очень похожи на указатели, но не требуют выполнять разыменование.

Вы можете вызвать эту функцию примерно следующим образом:

```
POLYGON2D poly;  // Считаем, что эта переменная корректно инициализирована

float min_x, min_y, max_x, max_y; // Используются для хранения результатов

// Вызов функции
Find_Bounding_Box_Poly2D(&poly, min_x, max_x, min_y, max_y);
```

Результатом работы функции является создание прямоугольника, данные которого будут сохранены в переменных (min_x, max_x, min_y, max_y). Используя эти переменные, а также координаты центра многоугольника (x_0, y_0), вы можете проводить проверку на столкновение объектов с использованием двух различных ограничивающих прямоугольников. Разумеется, эту проверку можно выполнить различными способами, например проверять, содержится ли какая-либо из четырех угловых точек одного прямоугольника внутри другого или использовать более интеллектуальные методы.

Содержание точки

Рассмотрим предложенный мною способ: проверим, не находится ли некоторая точка в пределах произвольного выпуклого многоугольника. Очевидно, что решение данной проблемы заключается в следующем.

Пусть прямоугольник задан точками (x_1, y_1) и (x_2, y_2) и мы хотим проверить, содержится ли в нем некоторая точка (x_0, y_0).

```
if (x0 >= x1 && x0 <= x2)  // Включение по X
  if (y0 >= y1 && y0 <= y2) // Включение по Y
    { /* Точка содержится в прямоугольнике */ }
```

НА ЗАМЕТКУ

Эти выражения можно объединить в один оператор if, но приведенный код с большей очевидностью демонстрирует, что значения координат вдоль осей X и Y могут обрабатываться независимо друг от друга.

Теперь рассмотрим, как можно узнать, находится ли некоторая точка внутри выпуклого многоугольника (как показано на рис. 8.41). На первый взгляд может показаться, что эта проблема достаточно проста, но уверяю вас, что это вовсе не так. Существует несколько способов ее решения, но один из самых простых — *проверка полупространств*. Если рассматриваемый многоугольник является выпуклым (что имеет место в нашем случае), можно считать каждую сторону отрезком, коллинеарным бесконечной плоскости, которая делит пространство на два полупространства (рис. 8.42).

Если рассматриваемая точка находится во внутренней стороне каждого из полупространств, то она содержится внутри многоугольника ввиду его выпуклости. Таким образом, все, что нужно сделать, — это каким-то образом проверить, находится ли точка двухмерного пространства по одну сторону от линии или по другую.

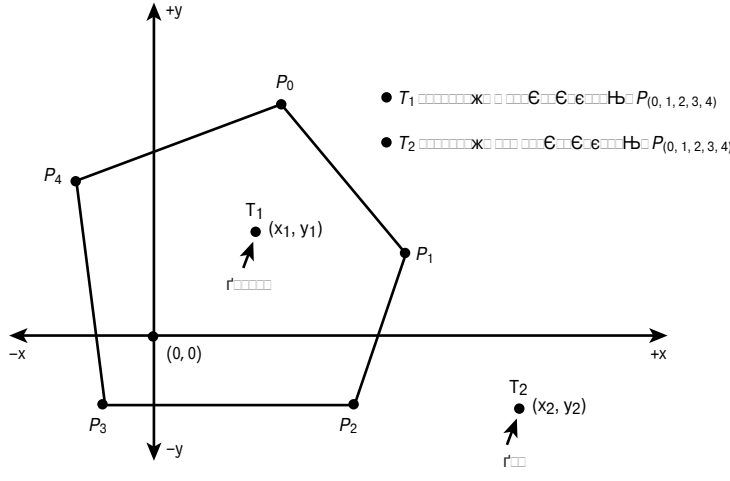


Рис. 8.41. Проверка содержания точки в многоугольнике

Если предположить, что линии помечаются в определенном порядке и преобразуются в векторы, это не так уж и сложно. Затем каждый из отрезков рассматривается как плоскость. Используя скалярное произведение, можно определить, находится ли точка по одну сторону от плоскости или по другую либо расположена непосредственно на самой плоскости.

Сейчас я хотел бы только одного: чтобы вы поняли геометрические идеи, положенные в основу такого решения, без рассмотрения самого алгоритма (который нужен будет в первую очередь при разработке трехмерных игр).

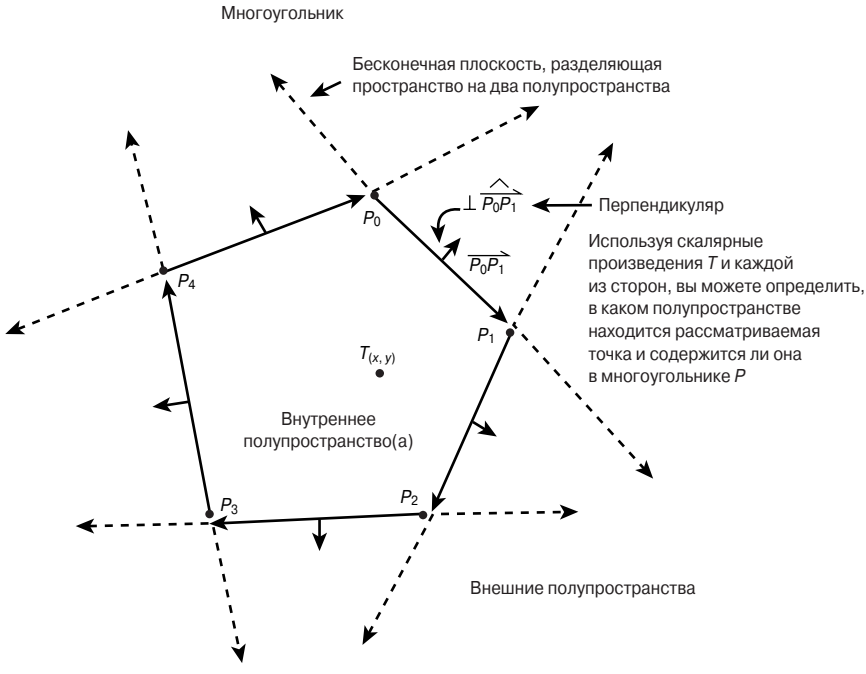


Рис. 8.42. Использование полупространств для решения задачи о точке внутри многоугольника

Немного о хронометрировании и синхронизации

До сих пор в большинстве программ я использовал очень примитивную синхронизирующую систему, основанную на функции Sleep(). В реальной жизни вы должны обеспечить определенную частоту кадров вашей игры, например 30 кадров в секунду. Чтобы решить эту задачу, лучше всего запустить таймер в начале игрового цикла (или отметить текущее время), а затем в конце игрового цикла проверить, соответствует ли истекший интервал времени частоте смены кадров, равной 30 кадров в секунду (т.е. он должен быть равен 1/30 секунды). Если да, переходим к следующему кадру. Если нет, ожидаем до тех пор, пока не пройдет нужное время (а пока можем работать над очередным кадром или выполнять некоторые служебные действия с тем, чтобы не расходовать зря драгоценное время).

В контексте исходного кода вы могли бы структурировать свою программу Game_Main() примерно таким образом:

```
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);

int Game_Main(void *parms = NULL, int num_parms = 0)
{
  // Вызывается для каждого кадра

  // Считываем текущее время в миллисекундах
```

```
  // с момента начала работы Windows
  Get _Clock();

  // Выполняем некоторые действия

  // Обеспечиваем частоту смены кадров, в данном
  // случае - 30 кадров в секунду
  Wait_Clock(30);

} // Game_Main
```

Все очень просто. В отношении использованных функций здесь можно сказать следующее: все они основаны на временны*х* функциях платформы Win32.

```
DWORD Get_Clock(void)
{
  // Функция возвращает текущее значение таймера
  return (GetTickCount());
} // Get_Clock

DWORD Start_Clock(void)
{
  // Функция запускает таймер, т.е. сохраняет текущее
  // показание времени. Используется в паре с функцией
  // Wait_Clock()

  return (start_clock_count = Get_Clock());
} // Start_Clock

DWORD Wait_Clock(DWORD count)
{
  // Функция выполняет ожидание в течение
  // определенного количества тактов, начиная с момента
  // вызова функции Start_Clock
  while ((Get_Clock() - start_clock_count) < count);
  return (Get_Clock());

} // Wait_Clock
```

Заметим, что в основе всех этих функций лежит функция GetTickCount(), входящая в состав API Win32, которая возвращает переменную типа DWORD, содержащую число миллисекунд, прошедших с момента запуска Windows. Следовательно, значения времени являются относительными! Глобальная переменная start_clock_count используется для сохранения времени начала отсчета. Она обновляется каждый раз при вызове функции Get_Clock() и определяется в библиотеке следующим образом:

```
DWORD start_clock_count = 0;
```

C++ Здесь можно смело применить классы C++. Попробуйте сделать это самостоятельно.

В DirectX есть также функция обнаружения вертикального обратного хода луча, которую можно использовать для определения состояния электронной пушки, когда она формирует изображение на электронно-лучевой трубке. Интерфейс IDIRECTDRAW4 поддерживает функцию, которая называется WaitForVerticalBlank(); ее прототип выглядит так:

```
HRESULT WaitForVerticalBlank(DWORD dwFlags,
                             HANDLE hEvent);
```

Вы можете использовать ее для ожидания различных состояний вертикального обратного хода луча. Флаги dwFlags управляют работой этой функции, а hEvent — это дескриптор события Win32 (для опытных программистов). В табл. 8.3 показаны возможные значения флагов.

Таблица 8.3. Значения флагов функции WaitForVerticalBlank ()

Флаг	*Описание*
DDWAITVB_BLOCKBEGIN	Возврат из функции в начале вертикального обратного хода луча
DDWAITVB_BLOCKEND	Возврат из функции по окончании вертикального обратного хода луча

Прокрутка и панорамирование

Я считаю прокрутку настолько простым делом, что практически никогда не описываю ее в своих книгах. В действительности же все игры, использующие прокрутку, образуют отдельный класс, и объяснение всех способов реализации прокрутки двухмерных изображений потребовало бы одной или двух частей книги. Я же хочу просто рассказать вам о разных способах организации прокрутки, а затем показать некоторые демонстрационные программы.

Подсистемы прокрутки страниц

В сущности *прокрутка страниц* означает, что по мере перемещения игрока по экрану и пересечения некоторой границы происходит полное обновление экрана и игрок как будто попадает в другую комнату. Эта технология реализуется очень легко и может кодироваться различными способами. На рис. 8.43 представлена типичная область игры, состоящая из 4×2 полных экранов размером 640×480 пикселей каждый. Следовательно, вся область игры охватывает 2560×960 пикселей.

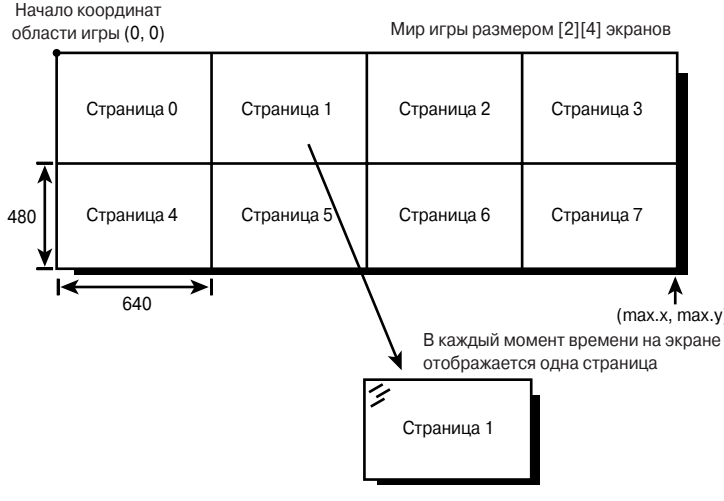


Рис. 8.43. Задание области прокрутки страниц

Для таких параметров игры логика отображения достаточно проста: вы загружаете первый экран в память, а все смежные экраны можно загрузить в оперативную память или создать для них на диске некоторую виртуальную область. Каким бы способом вы ее ни реализовали, прокрутка будет работать одинаково. По мере того как персонаж, руководимый игроком, перемещается по экрану, вы выполняете для него проверку некоторых граничных условий (вполне вероятно — достижение границы экрана). Если граничное условие удовлетворяется, вы переходите в следующую "комнату" или экран и помещаете героя в соответствующую позицию. Например, если вы двигаетесь слева направо и достигаете правой границы экрана, на нем должна показаться новая страница, на которой персонаж будет находиться уже слева. Не спорю, это весьма грубое представление, но это только начало.

Демонстрационный пример, иллюстрирующий работу этого метода — программа `DEMO8_10.EXE` на прилагаемом компакт-диске. Она создает область игры размером 3×1 и позволяет перемещать небольшую фигурку персонажа с помощью клавиш со стрелками. Если вы достигаете границы экрана, изображение обновляется. Я использовал растровые изображения, но не вижу никаких причин отказываться от векторных изображений или от сочетания тех и других.

Кроме того, в этой демонстрационной программе я немного схитрил, поскольку использовал функции из окончательного варианта библиотеки `T3DLIB1.CPP`, описанного в конце главы. Если захотите, вы всегда сможете взглянуть на исходный текст этих функций в указанном файле. Просто, чтобы сделать приличный пример, мне потребовалось больше возможностей, чем те, которые мы изучили до сих пор.

И наконец, обязательно взгляните на код перемещения по местности, содержащийся в этой демонстрационной программе. При перемещении персонаж постоянно опускается вниз, но по достижении определенного цвета, означающего землю, немного приподнимается, оставаясь таким образом все время над поверхностью земли.

Однородные подсистемы элементов мозаичного изображения

Рассмотренный пример нельзя считать настоящей прокруткой, которая происходит более плавно: полноэкранное изображение не обновляется постранично, а постепенно прокручивается вверх, вниз, вправо или влево.

DirectX позволяет добиться этого эффекта несколькими способами. Например, вы можете создать большую поверхность, а затем показывать на первичной поверхности экрана только ее часть (рис. 8.44).

Однако такой метод работает только при наличии DirectX 6.0 и выше и нуждается в аппаратном ускорении. Есть и более хороший подход, который состоит в том, что вы разбиваете ваш мир игры на элементы мозаичного изображения, а затем представляете каждый экран в виде квадратной матрицы, состоящей из элементов мозаичного изображения, т.е. клеток, содержащих растровое изображение, выводимое в данной позиции (рис. 8.45).

Например, вы можете задать для каждого такого элемента мозаичного изображения размер, равный 32×32 пикселя, и работать в режиме 640×480. Это означает, что один экран будет представлен мозаичной картой размером (640/32)×(480/32) = 20×15. Если ваш элемент мозаичного изображения будет иметь размеры 64×64, вам понадобится мозаичная карта размером (640/64)×(480/64) = 10×7.5 или, округляя в сторону уменьшения, 10×7 (7×64=448; последние 48 пикселей в нижней части экрана вы должны оставить для управляющей панели).

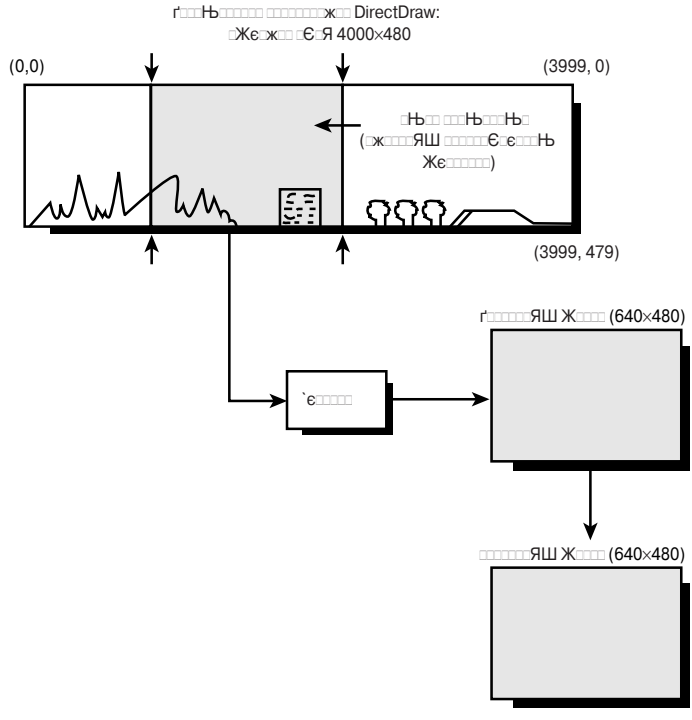


Рис. 8.44. Использование большой поверхности DirectDraw для достижения плавной прокрутки

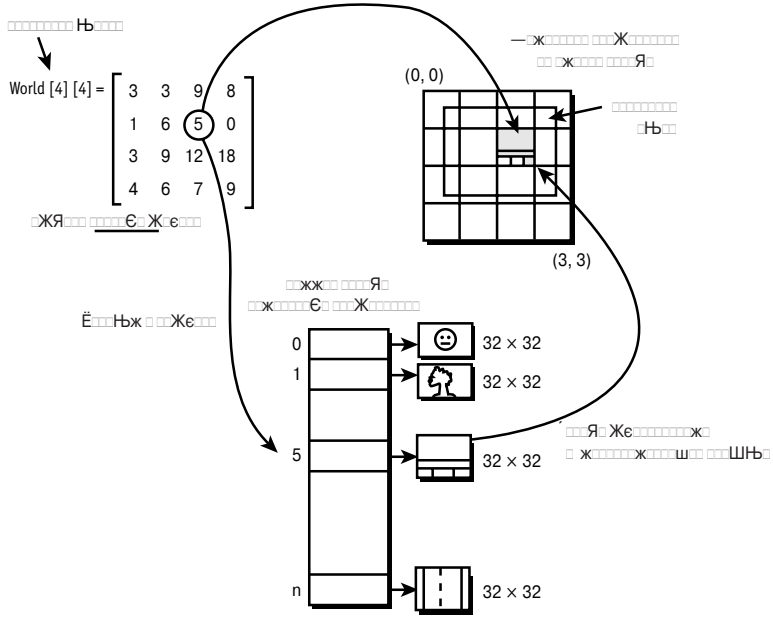


Рис. 8.45. Использование мозаичной структуры данных для представления прокручиваемого мира игры

Для решения этой задачи вам нужна соответствующая структура данных, например массив или матрица, содержащая целочисленные значения, или, возможно, структуры, в которых будет храниться информация о растровом изображении (в виде указателя или индекса), а также все остальные необходимые данные. Следующий фрагмент демонстрирует, каким образом можно создавать мозаичное изображение:

```
typedef struct TILE_TYP
{
  int x,y;    // Позиция элемента мозаичного изображения в матрице
  int index; // Индекс растрового изображения
  int flags;  // Флаги клетки

} TILE, *TILE_PTR;
```

Теперь, чтобы сохранить информацию об одном экране, вы должны использовать структуру наподобие приведенной ниже.

```
typedef struct TILED_IMAGE_TYP
{
  TILE image[7][10]; // 7 строк на 10 столбцов
} TILED_IMAGE, *TILE_IMAGE_PTR;
```

И наконец, вот мир игры, состоящий из 3×3 этих больших мозаичных изображений:

```
TILED_IMAGE world[3][3];
```

Вы можете также просто создать массив элементов мозаичного изображения, достаточный для хранения 3×3 экранов (т.е. массив размерностью 30×21), который будет выглядеть так:

```
typedef struct TILED_IMAGE_TYP
{
  TILE image[21][30]; // 21 строка на 30 столбцов
} TILED_IMAGE, *TILE_IMAGE_PTR;

TILED_IMAGE world;
```

НА ЗАМЕТКУ Вы можете спроектировать структуру данных любым способом, но с одним большим массивом работать легче, так как в этом случае не требуется переходить от одного экрана к другому по мере прокрутки очередной мозаичной карты (размером 10×7).

Итак, каким образом получается изображение каждого экрана? Вначале вам нужно загрузить свои растровые изображения в большой массив, состоящий из 64×64 поверхностей. У вас может быть один или несколько элементов мозаичного изображения, и некоторые их них могут повторяться, например корабли, обрывы, вода и т.д. На рис. 8.46 показан пример элементов мозаичного изображения.

Затем вы пишете сервисную программу или просто используете ASCII-редактор вместе с некоторым преобразующим программным обеспечением, что позволяет вам создавать мозаичные карты. Например, вы можете использовать данные в виде ASCII-кодов вместе с небольшим преобразующим инструментом таким образом, чтобы числа 0–9 указывали на элементы 0–9 набора элементов мозаичного изображения. Допустим, вы должны определить множество элементов мозаичного изображения, состоящее из 30×21 клеток. Я бы решал задачу следующим образом:

```
// Используем массив указателей на строки; можно было бы
// использовать массив символов или целочисленных значений,
// но его сложнее инициализировать.
```

```
// Символы 0-9 представляют битовые карты в
// памяти для хранения текстур
char* map1[21] =
{
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000",
"00000000000000000000000000000000"
};
```

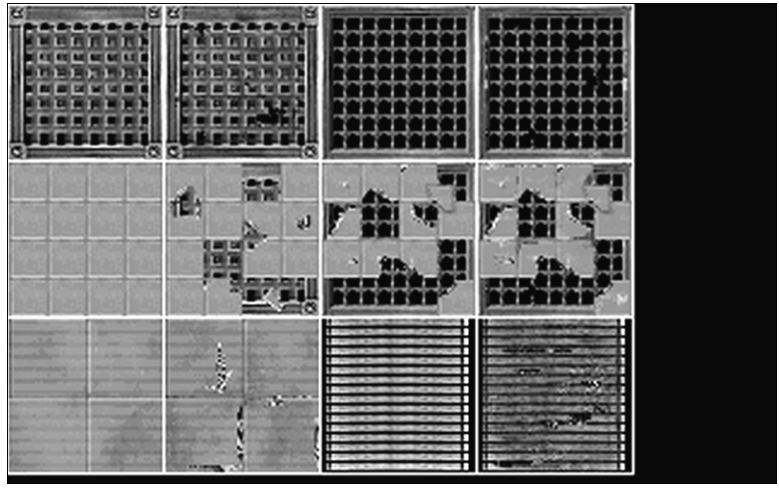


Рис. 8.46. Типичный набор мозаичных элементов

На стадии выполнения вы сканируете информацию, содержащуюся в карте, и помещаете ее в основную структуру, после чего вы готовы к выводу изображения. Чтобы вывести изображение, вначале вы должны настроить ткущее смотровое окно размером $m \times n$.

В большинстве случаев (но совсем не обязательно!) размеры этого окна совпадают с размерами экрана — 640×480, 800×600 и т.д. Справа у вас может располагаться управ-

ляющая панель или еще что-нибудь, что не должно прокручиваться. В любом случае, если предположить, что прокручивается весь экран, размеры которого составляют 640×480, вы должны принять во внимание два момента.

- В какой степени смотровое окно размером 640×480 перекрывает основную мозаичную карту, состоящую из 10×7 клеток.
- Граничные условия.

Теперь попытаемся понять, что я имею в виду. (Я думаю, здесь нужна помощь и мне самому, поскольку я окончательно запутался!) Допустим, смотровое окно находится в точке (0,0) в самом верхнем углу мозаичной карты (рис. 8.47).

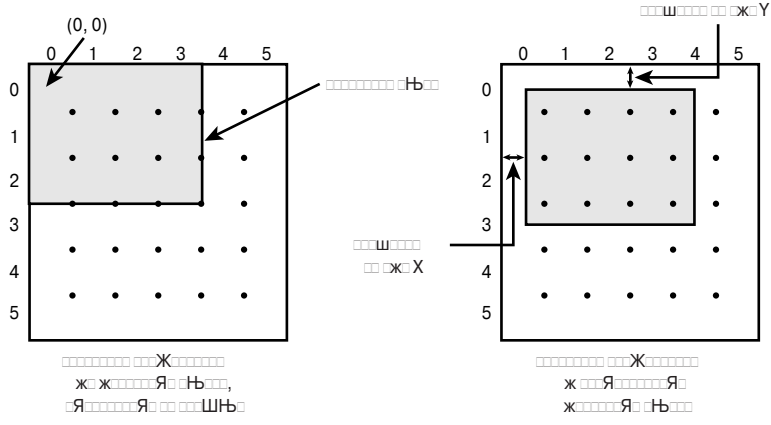


Рис. 8.47. Краевые проблемы, возникающие при прокрутке мозаичных карт

В данном случае вы должны нарисовать элементы мозаичного изображения, начиная от карты [0][0] и заканчивая картой [6][9]. Но вторая карта прокручивается только вправо или вниз, и вам нужно нарисовать некоторые граничные элементы мозаичной карты справа и непосредственно под текущим смотровым окном. Тогда при прокрутке одной целой клетки (размером 64×64) вам не придется рисовать всю строку и/или столбец мозаичной карты [0][0].

Очевидно, что каждый раз, когда вы пытаетесь нарисовать прямоугольный набор элементов мозаичного изображения (размеры которого всегда составляют 10×7), эти элементы будут поступать от одной или нескольких мозаичных карт. Более того, по мере прокручивания в ту или другую сторону от позиций, кратных 64, вы увидите только часть граничных элементов мозаичного изображения, поэтому здесь мы имеем дело с отсечением. К счастью, DirectX сама выполняет отсечение поверхностей растровых изображений, поэтому если вы рисуете растровое изображение, которое частично выходит за рамки экрана, оно просто будет отсечено. Это значит, что ваш окончательный алгоритм должен определять только отображаемые элементы мозаичного изображения, а также находить поверхности растровых изображений, представленные каждым из элементов мозаичного изображения, и затем передавать их блиттеру.

Чтобы проиллюстрировать все описанные действия, я написал демонстрационную программу DEMO8_11.CPP, которую вы можете найти на прилагаемом компакт-диске. Эта программа создает только что описанный мир игры и позволяет осуществлять в нем любые перемещения.

Подсистемы разреженных растровых мозаичных изображений

Единственная проблема, связанная с подсистемами мозаичных изображений, состоит в том, что существует огромное количество растровых изображений, которые нужно рисовать. Иногда вам хочется создать игру с прокруткой, но при этом не возиться с прокруткой огромного количества графической информации. Более того, порой вам не захочется делать все элементы мозаичного изображения одного размера. Именно так и получается в случае "космических стрелялок", поскольку подобные игры обычно используют огромное пустое пространство. Для игровых миров такого рода вам нужно создавать карту пространства, которая, как правило, бывает очень большой — например 4×4 (или 40×40) экранов. Тогда, вместо того чтобы хранить мозаичные карты для каждого экрана, вы просто помещаете каждый объект или растровое изображение в любое место в мировых координатах. При использовании этого метода любое растровое изображение объекта может не только иметь произвольные размеры, но и быть расположено в любом месте.

С этой схемой связана только одна проблема — эффективность. В сущности, для любой позиции, в которой в данный момент находится смотровое окно, вы должны находить все отображаемые объекты, которые оказываются внутри него. Если область игры невелика и объектов в ней немного, проверка приблизительно 100 объектов на предмет включения их в текущие окна не представляет большой проблемы. Но если вы должны протестировать тысячи объектов, это может навсегда отбить у вас охоту заниматься программированием игр...

Решение проблемы состоит в том, чтобы разделить всю область игры *на секторы*. По сути, вы создаете вспомогательную структуру данных, которая отслеживает все объекты и их связь с некоторыми клетками, на которые вы разделили всю область игры. Нельзя ли просто опять использовать набор элементов мозаичного изображения? И да, и нет. В данном случае секторы могут иметь произвольные размеры и на самом деле не быть каким-то образом связанными с размерами экрана. Выбор размеров секторов, скорее, имеет отношение к обнаружению столкновений и отслеживанию.

Обратимся к рис. 8.48, на котором показаны структуры данных и их связь с экраном, миром игры и смотровым окном.

Замечу, что это только один из способов решения данной проблемы, на самом деле их, безусловно, намного больше. Тем не менее суть состоит в том, что у вас есть определенное количество объектов, которые находятся на некотором расстоянии друг от друга в пространстве игры, размеры которой намного превышают размеры экрана (например, 100 000×100 000), и вы хотите иметь возможность перемещать объекты в этом пространстве. Для этого следует просто определить позиции всех объектов в координатах реального мира, а затем, исходя из места расположения окна просмотра (размеры которого составляют 640×480), отобразить все видимые объекты (разумеется, с надлежащим отсечением, поскольку многие объекты могут частично выходить за рамки поверхности экрана).

В качестве примера прокрутки разбросанных объектов я создал демонстрационную программу, которая позволяет вам перемещаться в звездном пространстве, заполненном рядом объектов. В этом примере нет больших структур данных, предназначенных для разбиения области игры на секторы, а сами объекты расположены случайным образом. В реальной жизни, конечно, имелись бы карты мира игры, разбитые на секторы, способствующие обнаружению столкновений и осуществлению оптимизации. Эта демонстрационная программа находится на прилагаемом компакт-диске в файле `DEMO8_12.CPP`. Она также использует библиотеку `T3DLIB1.CPP|H`, поэтому следует убедиться в том, что эти файлы включены в ваш проект.

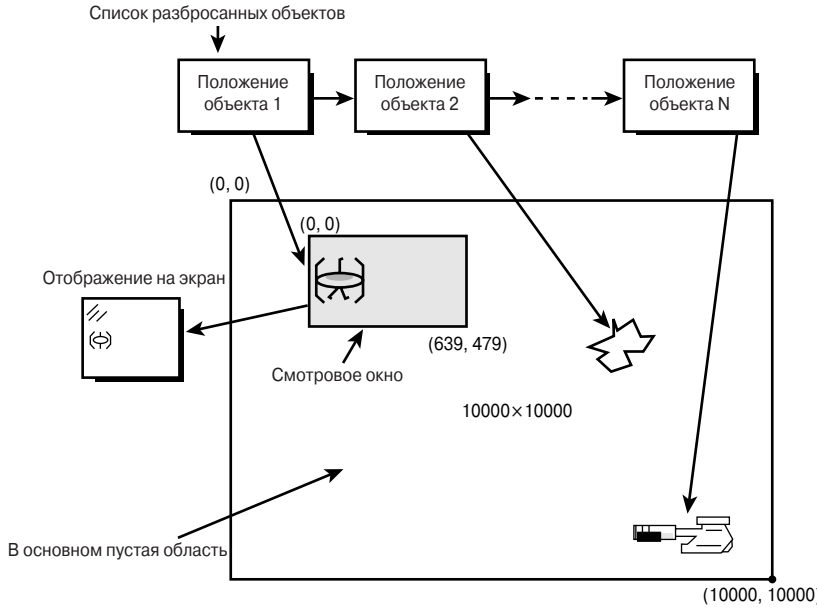


Рис. 8.48. Структуры данных, используемые в подсистеме прокрутки разбросанных объектов

Программа выполняет следующие основные действия: загружает ряд изображений объектов и случайным образом размещает их в пространстве, размеры которого составляют 10×10 экранов. Далее, как обычно, вы перемещаетесь по миру игры с помощью клавиш перемещения курсора. Вся прелесть прокрутки этого типа заключается в том, что здесь не должен отображаться весь экран: воспроизводятся только те растровые изображения, которые являются видимыми (или частично видимыми). Это значит, что в данном демонстрационном примере имеется фаза отсечения, в ходе которой проверяется каждый объект; и если он оказывается полностью невидимым, то просто не передается коду вывода на экран.

Псевдотрехмерные изометрические подсистемы

Должен признаться, что я получаю множество писем по электронной почте на эту тему, поэтому мне кажется, что пора садиться за работу над книгой под названием *Изометрические пространственные игры*! Вы спросите, что это такое? Отвечаю: это игры, в которых точка зрения игрока находится под некоторым углом (например, в 45°) к плоскости игрового мира. К таким изометрическим трехмерным играм можно отнести *Zaxxon*, *PaperBoy* и *Marble Madness*.

В наши дни изометрические игры снова входят в моду: *Diablo*, *Loaded* и множество других ролевых и военных компьютерных игр используют этот прием. Их популярность объясняется тем, что при таком подходе игра приобретает определенные черты трехмерной (при том, что создать изометрическую игру гораздо легче, чем настоящую трехмерную). Так как же создаются такие игры?

Вообще-то это секрет игрового сообщества, и я сомневаюсь, что вы найдете очень много информации на эту тему. Я же собираюсь предоставить вам некоторую пищу для ума и описать парочку способов, как это можно сделать. Небольших подсказок, которые

я намерен дать вам на страницах этой книги, должно быть более чем достаточно для того, чтобы вы самостоятельно смогли реализовать изометрическую подсистему.

СОВЕТ

Если вы действительно хотите изучить программирование трехмерных изометрических игр, то советую вам обратиться к книге *Isometric Game Programming with DirectX 7.0* (*Программирование изометрических игр при помощи DirectX 7.0*) Эрнеста Пазеры (Ernest Pazera).

Имеется три варианта решения этой задачи.

- Метод 1. Полностью двухмерная игра, основанная на ячейках.
- Метод 2. Полноэкранная игра, с двух- или трехмерными сетками столкновений.
- Метод 3. Игра с использованием трехмерной математики и с обзором фиксированной камерой.

Теперь рассмотрим каждый метод в отдельности.

Метод 1

При выборе этого метода вы должны принять решение об угле обзора и затем создать все изображения с учетом выбранного угла. Обычно все рисуется в виде прямоугольных мозаичных элементов, точно так же, как это делается в случае использования механизма прокрутки (например, рис. 8.49).

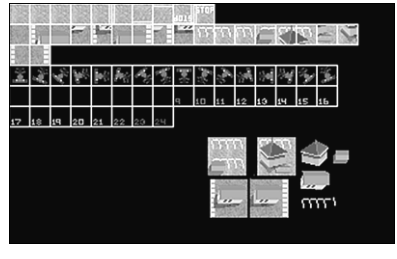


Рис. 8.49. Трехмерные изометрические заготовки

Основные сложности возникают при воспроизведении изображения. Если вы рисуете мир игры, то не можете просто нарисовать предметы в произвольном порядке. Вы должны нарисовать растровые изображения так, чтобы более отдаленные объекты были закрыты близлежащими. В сущности, вывод на экран похож на то, что делает художник, который начинает рисование с заднего плана и заканчивает передним. Это означает, что, когда вы задаете обзор под углом, то должны учитывать порядок расположения объектов при выводе кадра игры.

Это очень важно, в частности в том случае, когда у вас есть большой объект (например, дерево) и небольшая фигурка персонажа, который ходит за этим деревом. Это означает, что вы должны нарисовать его в строго определенный момент. Чтобы понять суть проблемы, рассмотрим рис. 8.50. Вам не придется ломать голову, если вы уверены, что нарисовали персонаж в нужный момент, т.е. после строки, которая расположена немного позади персонажа, или перед строкой, которая немного впереди него. Этот порядок рисования показан также на рис. 8.50. Следовательно, для рисования перемещаемых объектов в нужный момент вы должны выполнить определенные математические вычисления и сортировку.

Кроме того, вы можете использовать изометрическую подсистему с элементами мозаичного изображения разной высоты. Вы можете просто разместить сразу несколько элементов мозаики на нескольких строках или учитывать высоту элементов и рассматривать каждую строку мозаики как имеющую переменную высоту. Тогда при построении изо-

бражения вам иногда придется начинать отображение текущей строки с координаты Y, которая находится выше реальной позиции строки.

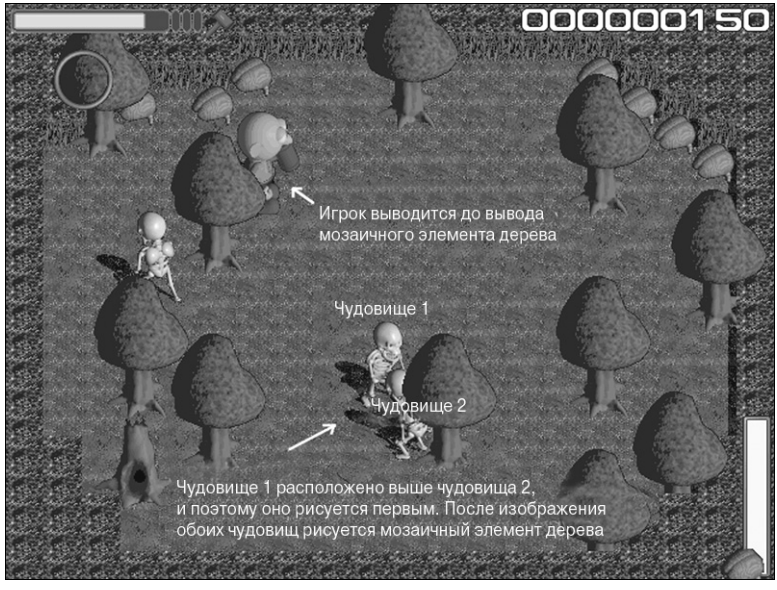


Рис. 8.50. При создании изометрических трехмерных изображений порядок объектов играет важную роль

На рис. 8.51 эта ситуация представлена графически. Сам процесс рисования остается без изменений, единственная проблема заключается в том, чтобы определить начальное значение координаты Y для каждой высоты блоков и добавить его к координате текущей строки (по сути, это просто высота ячейки мозаики).

А теперь немного усложним задачу. Пусть обзор задается углом наклона, равным 45°, и углом поворота, тоже равным 45°. Иными словами, мы имеем дело со стандартным обзором, принятым в играх *Diablo* и *Zaxxon*. В этом случае вам просто придется учитывать порядок объектов по оси X дополнительно к порядку по оси Y, а потому вы должны рисовать объекты слева направо (или справа налево, в зависимости от направления поворота) и сверху вниз. И вновь при рисовании вам следует упорядочивать свои объекты, но уже вдоль обеих осей — X и Y.

Это один из способов решения данной задачи. Разумеется, здесь имеется множество деталей, касающихся столкновений и т.п., и многие программисты, занимающиеся написанием игр, используют сложные шестиугольные и восьмиугольные системы координат, а затем отображают объекты в этих системах, но такой подход вы будете осваивать самостоятельно.

Метод 2

Полноэкранный метод намного интереснее, чем использование элементов мозаичного изображения. По сути, здесь вы рисуете некоторый изометрический трехмерный мир любым удобным для вас способом (например, с помощью программы моделирования объемных объектов или любых других, которые у вас имеются). Кроме того, вы можете использовать любые размеры экрана. Затем вы создаете вспомогательную структуру данных, которая содержит информацию о столкновениях, накладываемую на псевдодвухмерный мир игры. При этом подходе вы дополняете двухмерную информацию (которая не имеет данных ни о высоте, ни о

столкновениях) новыми данными, относящимися к двух- или трехмерному представлению (в зависимости от того, какой сложности вы хотите достичь).

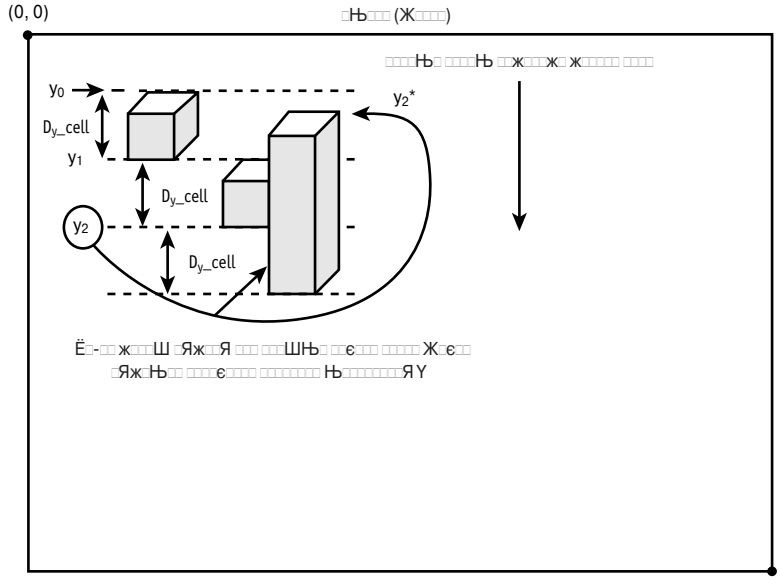


Рис. 8.51. Рисование больших клеток в изометрической трехмерной подсистеме

Затем вы просто рисуете все фоновое растровое изображение, но по мере вывода перемещаемых объектов выполняете отсечение. Эта технология показана на рис. 8.52, где представлено изометрическое изображение двухмерной сцены.

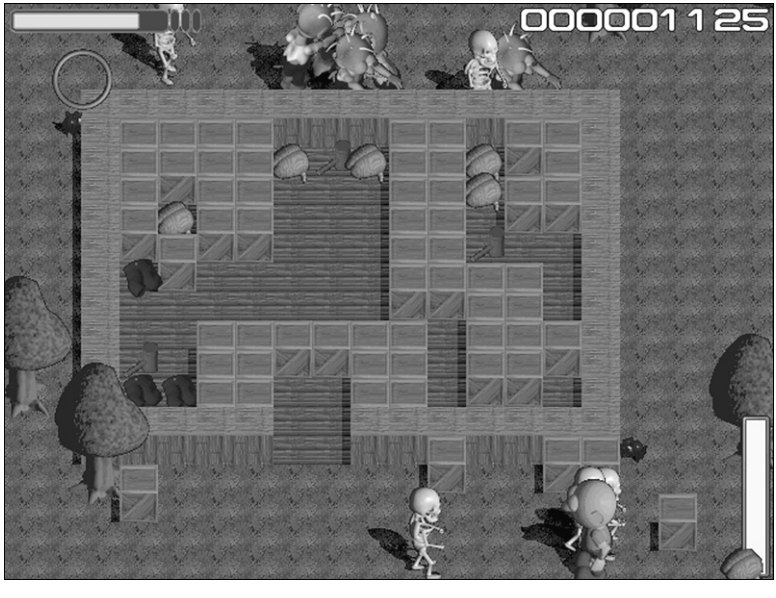


Рис. 8.52. Псевдотрехмерная изометрическая сцена

На рис. 8.53 вы видите ту же сцену с дополнительной информацией о многоугольниках, которая используется для реализации отсечения, столкновений и т.д.

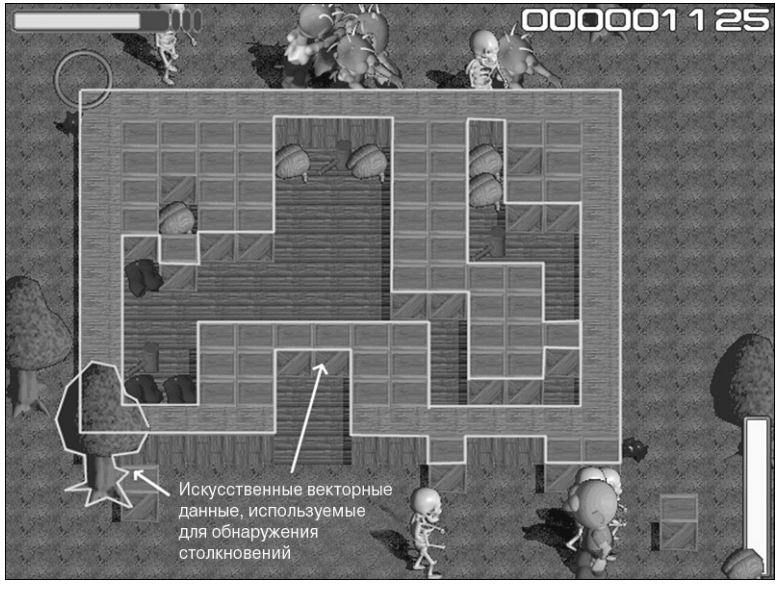


Рис. 8.53. Геометрические данные многоугольника, служащие для обнаружения столкновений, накладываются на псевдотрехмерное изображение

Чтобы сгенерировать эту сцену, вы должны сделать одно из двух: либо создать ее с помощью программного обеспечения для моделирования объемных деталей, либо разработать собственный инструментарий, который позволит вам точно отобразить информацию на каждом экране, содержащем различные данные. Я использовал оба метода, так что окончательный выбор за вами.

Метод 3

Это самый простой из всех методов, так как в нем не применяется никаких трюков. При этом используется настоящая трехмерная виртуальная машина, но вы просто фиксируете камеру так, чтобы получить изомерный обзор, — на этом изометрическая игра готова. Более того, поскольку вам известно, что обзор всегда осуществляется под определенным углом, вы можете несколько оптимизировать порядок вычерчивания и сложность сцены. Именно так и работают многие изометрические игры, предназначенные для игровых приставок Sony PlayStation I и II; это настоящие трехмерные игры, но их обзор ограничен углом наклона в 45°.

НА ЗАМЕТКУ

Рассматриваемая тема очень велика и вполне может быть выделена в отдельную небольшую книгу, так что если приведенных намеков вам недостаточно, то дополнительный материал на эту тему вы сможете найти на прилагаемом компакт-диске.

Библиотека T3DLIB1

Теперь можно приступить к рассмотрению всех определений, макросов, структур данных и функций, которые были созданы по ходу чтения. Я разместил их в двух фай-

лах — T3DLIB1.CPP|H. Вы можете включать их в свои программы и использовать всю изученную к настоящему моменту информацию, не затрачивая усилий на поиски этого кода в многочисленных демонстрационных программах.

Помимо прочего, для упрощения программирования двухмерных игр я создал несколько дополнительных функций для двухмерных спрайтов, которые использовал при создании демонстрационных программ. На подробные объяснения я не намерен тратить много времени, но моих комментариев будет вполне достаточно, чтобы понять, что и как работает. А теперь рассмотрим каждый фрагмент кода.

Архитектура графической подсистемы

До этого момента мы рассматривали достаточно простую двухмерную подсистему, схема которой показана на рис. 8.54. По сути, это виртуальная машина DirectX для работы с двухмерной графикой в 8- и 16-битовых режимах с использованием вторичных буферов. В ней поддерживается любое разрешение, отсечение первичной поверхностью экрана и возможность использования прозрачности в оконном режиме.

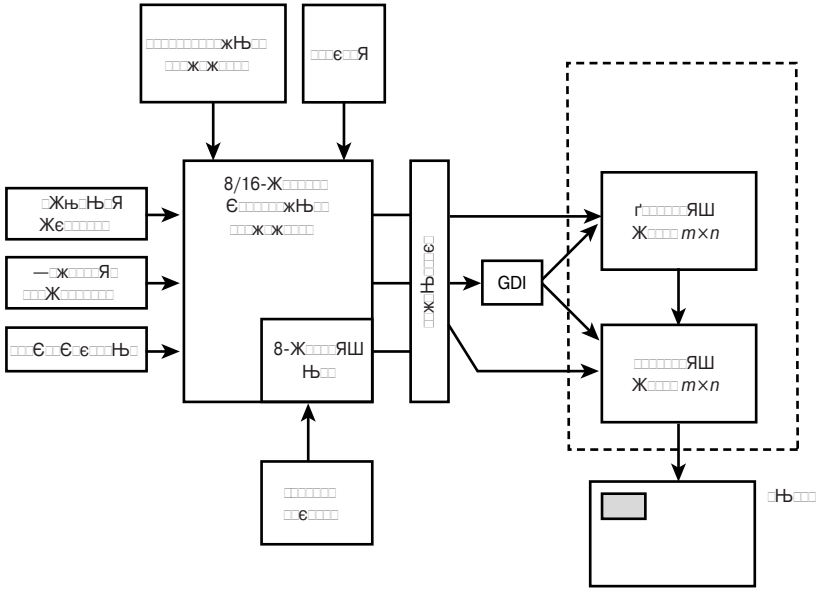


Рис. 8.54. Архитектура графической подсистемы

Чтобы создать приложение, использующее данную библиотеку, вы должны включить в проект файлы T3DLIB1.CPP|H (они находятся на прилагаемом компакт-диске), а также библиотеки DDRAW.LIB, WINMM.LIB и основной файл игры T3DCONSOLE2.CPP (он является расширенной версией более простой программы T3DCONSOLE.CPP, которая уже использовалась ранее в этой книге).

T3DCONSOLE2.CPP: новая консоль для программирования игр

Прежде чем рассматривать все функциональные возможности библиотеки, познакомимся вкратце с файлом T3DCONSOLE2.CPP, в котором теперь имеется поддержка 8- и 16-битовых оконных и полноэкранных режимов, а также ряд новых возможностей. Изменив всего лишь несколько директив препроцессора #define в самом начале файла, вы сможете

выбрать 8- или 16-битовый, полноэкранный или оконный режимы. Вот код, содержащийся в рассматриваемом файле.

```
// T3DCONSOLE2.CPP -
// если хотите, можете использовать его в качестве шаблона
// для своих приложений. Изменять можно любые
// характеристики, например разрешение, оконный
// или полноэкранный режим, устройства ввода
// DirectInput и так далее.
// В настоящий момент приложение создает экран 640x480x16
// в оконном режиме, следовательно, перед запуском
// приложения вы должны находиться в режиме использования
// 16-битового цвета. Переход в полноэкранный режим
// осуществляется очень просто: нужно заменить на FALSE (0)
// значение константы WINDOWED_APP, определяемой директивой
// препроцессора #define. Аналогично для задания
// глубины цвета измените соответствующую
// константу в функции DDraw_Init() (эту функцию вызывает
// функция Game_Init())

// ОБЯЗАТЕЛЬНО ПРОЧИТАЙТЕ ЭТОТ КОММЕНТАРИЙ!
// Перед компиляцией убедитесь в том, что в
// проект включены библиотеки: DDRAW_LIB,
// DSOUND_LIB, DINPUT_LIB и WINMM.LIB, а также модули
// библиотеки T3D: T3DLIB1.CPP, T3DLIB2.CPP
// и T3DLIB3.CPP; кроме того, в рабочем каталоге
// должны находиться файлы заголовков T3DLIB1.H,
// T3DLIB2.H и T3DLIB3.H

////////////// ВКЛЮЧАЕМЫЕ ФАЙЛЫ ////////////
#define  INUITGUID  // обеспечиваем доступность всех
            // COM-интерфейсов. Вместо этого
            // вы можете включить библиотечный
            // файл DXGUID.LIB

#define  WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>    // Заголовочные файлы DirectX
#include <dsound.h>
#include <dmksctrl.h>
```

```
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h"  // Заголовочные файлы библиотеки T3D
#include "T3DLIB2.h"
#include "T3DLIB3.h"

////////////  ОПРЕДЕЛЕНИЯ  ////////////

// Определения интерфейса окна
#define  WINDOW_CLASS_NAME  "WIN3DCLASS"  // Имя класса
#define  WINDOW_TITLE      "T3D Graphic Console Ver 2.0"
#define  WINDOW_WIDTH      640                    // Размеры окна
#define  WINDOW_HEIGHT     480
#define  WINDOW_BPP        16                      // Глубина цвета

// Примечание: в оконном режиме
// значение глубины цвета должно совпадать с
// соответствующей системной величиной, кроме
// того, в случае 8-битовой глубины цвета
// создается и подключается палитра цветов

#define  WINDOWED_APP     1 // 0 - полноэкранный и 1 - оконный режимы


////////////  ПРОТОТИПЫ ФУНКЦИЙ ////////////
int Game_Init (void *parms = NULL);
int Game_Shutdown (void *parms = NULL);
int Game_Main (void *parms = NULL);

////////////  ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////
HWND  main_window_handle = NULL; // Дескриптор окна
HINSTANCE  main_instance = NULL;   // Экземпляр
char buffer [256];                 // Используется для вывода текста

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

  // Сообщение
  switch (msg)
  {
    case WM_CREATE:
      {
        // Выполняем инициализацию
        return (0);
      }  break;
```

```
    case WM_PAINT:
      {
        // Закрашивание
        hdc = BeginPaint (hwnd, &ps);

        // Конец закрашивания
        EndPaint (hwnd, &pc);
        return(0);
      } break;

    case WM_DESTROY:
      {
        // Закрываем приложение
        PostQuitMessage(0);
        return(0);
      } break;

    default: break;

  } // switch

  // Обрабатываем все неучтенные сообщения
  return (DefWindowProc (hwnd, msg, wparam, lparam));

} // WinProc

//////////// WINMAIN ////////////
int WINAPI WinMain (HINSTANCE hinstance,
            HINSTANCE hprevinstance,
            LPSTR     lpcmdline,
            int       ncmdshow)
{
  WNDCLASSEX winclass;   // Класс, который мы создаем
  HWND       hwnd;       // Дескриптор окна
  MSG        msg;        // Стандартное сообщение
  HDC        hdc;        // контекст устройства

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

```
// Сохраняем hinstance в глобальной переменной
main_instance = hinstance;

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

// Создаем окно
if (!(hwnd = CreateWindowEx (NULL,
                     WINDOW_CLASS_NAME,
                     WINDOW_TITLE,
                     WINDOWED_APP ?
                     (WS_OVERLAPPED|WS_SYSMENU|
                      WS_VISIBLE):
                     (WS_POPUP|WS_VISIBLE)),
    0,0,  // Начальное положение
    WINDOW_WIDTH, WINDOW_HEIGHT,
    NULL,        // Дескриптор родительского окна
    NULL,        // Дескриптор меню
    hinstance,   // Экземпляр данного приложения
    NULL ))      // Дополнительный параметр
   return (0);

// Сохраняем дескриптор окна и экземпляр в
// глобальных переменных
main_window_handle = hwnd;
main_instance      = hinstance;

// Изменяем размеры окна с тем, чтобы размеры клиентской
// области действительно равнялись произведению ширины
// на высоту
if (WINDOWED_APP)
{
   // Изменяем размеры окна, чтобы клиентская область
   // имела реальные заданные размеры; с этого момента,
   // если приложение работает в оконном режиме, могут
   // появиться границы и элементы управления; если
   // приложение полноэкранное, то все это не имеет
   // значения
   RECT window_rect = { 0, 0, WINDOW_WIDTH,
                        WINDOW_HEIGHT};

   // Настройка рамок окна
   AdjustWindowRectEx(
         &window_rect,
         GetWindowStyle(main_window_handle),
         GetMenu(main_window_handle)!=NULL,
         GetWindowExStyle(main_window_handle));

   // Сохраняем глобальные смещения клиентской области
   window_client_x0 = -window_rect.left;
   window_client_y0 = -window_rect.top;

   // Изменяем размеры окна при помощи функции
   // MoveWindow()
```

```
    MoveWindow(main_window_handle,
          CW_USEDEFAULT, //  Значение координаты x
          CW_USEDEFAULT, //  Значение координаты y
          window_rect.right  - window_rect.left,
          window_rect.bottom - window_rect.top,
          TRUE);

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

  // Выполняем всю необходимую
  // инициализацию игровой консоли
  Game_Init();

  // Вход в основной цикл событий
  while(1)
  {
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
      // Проверяем условие выхода
      if (msg.message == WM_QUIT)
        break;
      // Обрабатываем нажатие клавиш быстрого вызова
      TranslateMessage (&msg);

      // Передаем сообщение процедуре окна
      DispatchMessage (&msg);
    } // if

    // Основная работа игры
    Game_Main();

  } // while

  // Закрываем игру и освобождаем все ресурсы
  Game_Shutdown();

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

} // WinMain

////////////// ФУНКЦИИ T3D II ////////////
int Game_Init(void *parms)
{
  // В этой функции вы выполняете всю инициализацию,
  // необходимую для вашей игры

  // Запускаем DirectDraw (в указанных параметрах можно
  // передавать нужные значения)
  DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT,
        WINDOW_BPP, WINDOWED_APP);
```

```
  // Инициализируем подсистему DirectInput
  DInput_Init();

  // Получаем клавиатуру
  DInput_Init_Keyboard();

  // Вызываем другие функции для получения остальных
  // устройств DirectInput...

  // Инициализируем подсистемы DirectSound и DirectMusic
  DSound_Init();
  DMusic_Init();

  // Делаем курсор мыши невидимым
  ShowCursor(FALSE);

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

  // Здесь должен находиться весь остальной код,
  // выполняющий инициализацию...

  // Возвращаем код успешного завершения функции
  return (1);

} // Game_Init

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

int Game_Shutdown(void *parms)
{
  // Эта функция служит для закрытия вашей игры и
  // освобождения всех выделенных ресурсов

  // Закрываем все

  // Освобождаем все ресурсы, выделенные во время игры...

  // Теперь закрываем подсистему DirectSound
  DSound_Stop_All_Sounds();
  DSound_Delete_All_Sounds();
  DSound_Shutdown();

  // и подсистему DirectMusic
  DMusic_Delete_All_MIDI ();
  DMusic_Shutdown();

  // и только теперь закрываем подсистему DirectInput
  DInput_Release_Keyboard();
  DInput_Shutdown();

  // Самой последней закрываем подсистему DirectDraw
  DDraw_Shutdown();
```

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

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

int Game_Main ( void *parms )
{
  // "рабочая лошадка" вашей игры; постоянно
  // вызывается в реальном времени и похожа на
  // функцию main() языка C; все обращения к функциям,
  // которые вы делаете в своей игре, осуществляются
  // именно в ней

  int index;  // Переменная цикла

  // Запускаем таймер
  Start_Clock();

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

  // Считываем данные клавиатуры и других устройств
  DInput_Read_Keyboard();

  // Здесь находится логика игры...

  // Переключаем поверхности
  DDraw_Flip();

  // Осуществляем синхронизацию для достижения
  // частоты кадров, равной 30 кадрам в секунду
  Wait_Clock(30);

  // Проверяем, не пытается ли пользователь выйти
  if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
    PostMessage (main_window_handle, WM_DESTROY, 0, 0);
  } // if

  // Возвращаем код успешного завершения
  return(1);

} // Game_Main
```

Вы можете изменить значения следующих констант, задаваемых директивой препроцессора #define:

```
#define WINDOW_WIDTH    640   // Размеры окна
#define WINDOW_HEIGHT   480
#define WINDOW_BPP      16    // Глубина цвета

// Примечание: в оконном режиме
// значение глубины цвета должно совпадать с
// соответствующей системной величиной, кроме
```

```
// того, в случае 8-битовой глубины цвета
// создается и подключается палитра цветов

#define WINDOWED_APP    1 // 0 - полноэкранный и 1 - оконный режимы
```

Это позволит самостоятельно задавать разрешение экрана для полноэкранных режимов, глубину цвета полноэкранных режимов или использовать оконный режим (правда, в последнем случае вы можете регулировать только размеры окна, поскольку в оконном режиме используется текущее значение глубины цвета и текущее значение разрешения экрана).

Основные определения

Библиотека имеет один заголовочный файл T3DLIB1.H, в котором среди прочего содержится ряд используемых в ней макроопределений. Ниже приведены основные из них; полностью с макроопределениями вы можете ознакомиться, обратившись к файлу на прилагаемом компакт-диске.

```
////////////// ОПРЕДЕЛЕНИЯ //////////////
// Значения по умолчанию для экрана; все они могут быть
// изменены в результате вызова функции DDraw_Init()
#define SCREEN_WITH    640 // Размеры экрана
#define SCREEN_HEIGHT  480
#define SCREEN_BPP     8    // Глубина цвета
#define MAX_COLORS_WITH 256

#define DEFAULT_PALETTE_FILE "PALDATA2.PAL"

// Режим игры - полноэкранный/оконный
#define SCREEN_FULLSCREEN 0
#define SCREEN_WINDOWED  1

// Определения для работы с .BMP-файлами
#define BITMAP_ID     0x4D42 // идентификатор
#define BITMAP_STATE_DEAD   0
#define BITMAP_STATE_ALIVE  1
#define BITMAP_STATE_DYING  2
#define BITMAP_ATTR_LOADED 128

#define BITMAP_EXTRACT_MODE_CELL  0
#define BITMAP_EXTRACT_MODE_ABS  1

// Форматы пикселей DirectDraw
#define DD_PIXEL_FORMAT8      8
#define DD_PIXEL_FORMAT555    15
#define DD_PIXEL_FORMAT565    16
#define DD_PIXEL_FORMAT888    24
#define DD_PIXEL_FORMATALPHA888 32

// Определения числа PI
#define PI      ((float)3.141592654f)
#define PI2     ((float)6.283185307f)
#define PI_DIV_2  ((float)1.570796327f)
#define PI_DIV_4  ((float)0.785398163f)
#define PI_INV    ((float)0.318309886f)
```

```
// Константы для математики с фиксированной точкой
#define FIXP16_SHIFT    16
#define FIXP16_MAG      65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff000
#define FIXP16_ROUND_UP 0x00008000
```

В том или ином месте вы должны были встречаться со всеми этими данными.

Рабочие макросы

Ниже приводятся макросы, разработанные к данному моменту. С ними вы тоже уже встречались, а здесь они просто собраны в одном месте.

```
// Макросы считывания данных клавиатуры в
// асинхронном режиме
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code)\
                  & 0x8000)? 1 : 0)
#define KEY_UP(vk_code)   ((GetAsyncKeyState(vk_code)\
                  & 0x8000)? 0 : 1)

// Макрос создает 16-битовое значение цвета в формате
// 5.5.5 (1-битовый альфа-режим)
#define _RGB16BIT555(r,g,b) ((b&31)+((g&31)<<5)+((r&31)<<10))

// Макрос создает 16-битовое значение цвета в формате
// 5.6.5 (режим с доминированием зеленого цвета)
#define _RGB16BIT565(r,g,b) ((b&31)+((g&63)<<5)+((r&31)<<11))

// Макрос создает 24-битовое значение цвета в формате
// 8.8.8
#define _RGB24BIT(a,r,g,b) ((b)+((g)<<8)+((r)<<16))

// Макрос создает 32-битовое значение цвета в формате
// A.8.8.8 (8-битовый альфа-режим)
#define _RGB32BIT(a,r,g,b) ((b)+((g)<<8)+((r)<<16)+((a)<<24))

// Макросы, выполняющие поразрядные операции
#define SET_BIT(word,bit_flag)   ((word)=((word)|(bit_flag)))
#define RESET_BIT(word,bit_flag) ((word)=((word)&(~bit_flag)))

// Инициализация структуры DirectDraw
#define DDRAW_INIT_STRUCT(ddstruct) \
     { memset(&ddstruct,0,sizeof(ddstruct)); \
       ddstruct.dwSize = sizeof(ddstruct); }

// Макросы для вычисления большего и меньшего из двух выражений
#define  MIN(a,b)  (((a) < (b)) ? (a) : (b))
#define  MAX(a,b)  (((a) > (b)) ? (b) : (b))

// Макрос перестановки
#define SWAP(a,b,t) { t=a; a=b; b=t; }
```

```
// Макросы некоторых математических операций
#define DEG_TO_RAD(ang)   ((ang)*PI/180.0)
#define RAD_TO_DEG(rads)  ((rads)*180.0/PI)

#define RAND_RANGE(x,y) ((x)+(rand()%((y)-(x)+1)))
```

Структуры и типы данных

Следующие фрагменты кода содержат структуры и типы данных, используемые библиотекой. Я собираюсь показать здесь весь список, но будьте внимательны, поскольку с некоторыми из них вы еще не знакомы (они используются с объектами блиттера). Чтобы не нарушать последовательности изложения, далее приведены все типы данных библиотеки.

```
// Основные беззнаковые типы
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char  UCHAR;
typedef unsigned char  BYTE;
typedef unsigned int   QUAD;
typedef unsigned int   UINT;

// Контейнерная структура для BMP-файла
typedef struct  BITMAP_FILE_TAG
{
  BITMAPFILEHEADER bitmapfileheader;
    // Заголовок файла с растровым изображением
  BITMAPINFOHEADER bitmapinfoheader;
    // Вся информация, включая данные палитры
  PALETTEENTRY    palette[256];
    // Палитра
  UCHAR*          buffer;
    // Указатель на буфер данных
} BITMAP_FILE, *BITMAP_FILE_PTR;

// Структура объекта блиттера
typedef struct BOB_TYP
{
  int state;            // Состояние объекта
  int anim_state;    // Состояние анимации,
                        // заполняется по вашему усмотрению
  int attr;             // Атрибуты объекта
  float x,y;            // Позиция, в которой будет выведено
                        // растровое изображение
  float xv, yv;         // Скорость объекта
  int width, height; // Ширина и высота объекта блиттера
  int width_fill;     // Внутренняя переменная, используемая
                        // для увеличения ширины поверхности
  int counter_1;     // Общие счетчики
  int counter_2;
  int max_count_1;   // Основные пороговые значения;
  int max_count_2;
  int varsI[16];        // Стек для 16 целых чисел
  int varsF[16];        // и для 16 чисел с плавающей точкой
```

```
  int curr_frame;        // Текущий кадр анимации
  int num_frames;        // Полное число кадров анимации
  int curr_animation;  // Индекс текущего кадра анимации
  int anim_counter;    // Используется для хронометрирования
                        // изменений анимации
  int anim_index;       // Индекс элемента анимации
  int anim_count_max;// Количество циклов, предшествующих
            // анимации
  int *animations[MAX_BOB_ANIMATIONS];
            // Анимационные последовательности
  LPDIRECTDRAWSURFACE7  images[MAX_BOB_FRAMES];
    // Растровые изображения поверхностей DirectDraw
} BOB,  *BOB_PTR;

// Простое растровое изображение
typedef struct BITMAP_IMAGE_TYP
{
  int state;            // Состояние растрового изображения
  int attr;             // Атрибуты растрового изображения
  int x,y;              // Позиция растрового изображения
  int width, height;  // Размеры растрового изображения
  int num_bytes;     // Полное число байт растрового
                        // изображения
  UCHAR*  buffer;    // Пиксели растрового изображения
}  BITMAP_IMAGE,  *BITMAP_IMAGE_PTR;

// Структура, описывающая мигающий свет
typedef struct BLINKER_TYP
{
  // эти поля задаются пользователем
  int color_index;            // Индекс мигающего цвета
  PALETTEENTRY on_color; // Значение "включенного" цвета
  PALETTEENTRY on_color; // Значение "выключенного" цвета
  int on_time;                 // Число "включенных" кадров
  int on_time;                 // Число "выключенных" кадров

  // Внутренние члены структуры
  int counter;    // Счетчик изменений состояний
  int state;       // Состояние света: -1 - выключен, 1 - включен, 0-неактивен
} BLINKER, *BLINKER_PTR;

// 2D-вершина
typedef struct VERTEX2DI_TYP
{
  int x,y;
} VERTEX2DI,  *VERTEX2DI_PTR;

// 2D-вершина
typedef struct VERTEX2DF_TYP
{
  float x,y;
} VERTEX2DF, *VERTEX2DF_PTR;
```

```
// 2D-многоугольник
typedef struct POLYGON2D_TYP
{
   int state;             // Состояние многоугольника
   int num_verts;     // Число вершин
   int x0, y0;             // Координаты центра многоугольника
   int xv, yv;             // Начальная скорость
   DWORD color;       // Индекс или PALETTEENTRY
   VERTEX2DF* vlist; // Указатель на список вершин
} POLYGON2D, *POLYGON2D_PTR

// Матрица 3x3
typedef struct MATRIX3X3_TYP
{
   union
   {
      float M[3][3];   // массив для данных
      // Упорядоченные и явно обозначенные переменные
      struct
      {
         float M00, M01, M02;
         float M10, M11, M12;
         float M20, M21, M12;
      };
   }; // union
} MATRIX3X3,  *MATRIX3X3_PTR;

// Матрица 1x3
typedef struct MATRIX1X3_TYP
{
   union
   {
      float M[3]; // Массив данных
      // Упорядоченные и явно обозначенные переменные
      struct
      {
         float M00, M01, M02;
      } ;
   }; // union
} MATRIX1X3,  *MATRIX1X3_PTR;

// Матрица 3x2
typedef struct MATRIX3X2_TYP
{
   union
   {
      float M[3][2]; // Массив данных
      // Упорядоченные и явно обозначенные переменные
      struct
      {
         float M00, M01;
         float M10, M11;
         float M20, M21;
```

```
    } ;
  }; // union
} MATRIX3X2, *MATRIX3X2_PTR;

// Матрица 1x2
typedef struct MATRIX1X2_TYP
{
  union
  {
    float M[2]; // Массив данных
    // Упорядоченные и явно обозначенные переменные
    struct
    {
      float M00, M01;
    } ;
  }; // union
} MATRIX1X2, *MATRIX1X2_PTR;
```

В сущности, ничего нового здесь нет: основные типы, информация о растровом изображении, данные о многоугольнике и немного матричной математики.

Глобальные переменные

Вы, наверное, заметили, что я предпочитаю использовать глобальные переменные, поскольку работать с ними быстрее, чем с локальными переменными. Кроме того, они хорошо справляются с ролью переменных системного уровня (которые в большом количестве присутствуют в двух- и трехмерных подсистемах). Ниже приведены глобальные переменные, используемые в нашей библиотеке. Здесь тоже многие переменные покажутся вам знакомыми; остальные снабжены комментариями, которые помогут разобраться в их назначении.

```
extern FILE *fp_error;              // Основной файл ошибок
extern char  error_filename[80]; // Имя файла ошибок

// Обратите внимание, что ряд интерфейсов использует
// интерфейс версии 4.0
extern  LPDIRECTDRAW7      lpdd;
            // Объект DirectDraw
extern  LPDIRECTDRAWSURFACE7 lpddsprimary;
            // Первичная поверхность DirectDraw
extern  LPDIRECTDRAWSURFACE7 lpddsback;
            // Вторичная поверхность DirectDraw
extern  LPDIRECTDRAWPALETTE  lpddpal;
            // Палитра DirectDraw
 extern  LPDIRECTDRAWCLIPPER lpddclipper;
            // Отсекатель DirectDraw
            // для вторичной поверхности
extern  LPDIRECTDRAWCLIPPER  lpddclipperwin;
            // Отсекатель DirectDraw,
            // используемый для окна
extern  PALETTEENTRY   palette[256];
            // Цветовая палитра
extern  PALETTEENTRY   save_palette[256];
            // Используется для сохранения палитр
extern  DDSURFACEDESC2 ddsd;
```

```
            // Описание поверхности DirectDraw
extern  DDBLTFX      ddbltfx;
            // Используется для заливки
extern  DDSCAP32     ddscaps;
            // Описание поверхности DirectDraw
extern  HRESULT      ddrval;
            // Результат обращений к DirectDraw
extern UCHAR* primary_buffer;    // Первичный видеобуфер
extern UCHAR* back_buffer;       // Вторичный видеобуфер
extern int    primary_lpitch;    // Шаг видеопамяти
extern int    back_lpitch;       // Шаг видеопамяти
extern BITMAP_FILE bitmap8bit;  // Файл с 8-битовым
                    // растровым изображением
extern BITMAP_FILE bitmap16bit; // Файл с 16-битовым
                    // растровым изображением
extern BITMAP_FILE bitmap24bit; // Файл с 24-битовым
                    // растровым изображением

extern DWORD  start_clock_count;  // Используется для
                     // хронометрирования
extern int    windowed_mode;      // Отслеживает режим
                     // работы DirectDraw

// Основной отсекающий прямоугольник
extern int min_clip_x,
       max_clip_x,
       min_clip_y,
       max_clip_y;

// Эти переменные будут замещены в результате
// вызова DD_Init()
extern  int  screen_width,   // Ширина экрана
        screen_height,       // Высота экрана
        screen_bpp,          // Число бит на пиксель
        screen_windowed;  // Тип приложения

extern  int  dd_pixel_format; // формат пикселей по
        // умолчанию, заданный вызовом DDraw_Init()

extern int  window_client_x0;
extern int  window_client_y0;
            // Используются для отслеживания
            // начальных значений координат
            // клиентской области во время
            // функционирования в оконном режиме

// Таблицы поиска
extern  float cos_look[361];
extern  float sin_look[361];

// Указатель и функции построения RGB-цветов
extern  USHORT (*RGB16Bit)(int r, int g, int b);
extern  USHORT RGB16Bit565(int r, int g, int b);
extern  USHORT RGB16Bit555(int r, int g, int b);
```

Интерфейс DirectDraw

Теперь, когда вы познакомились со всеми типами данных и переменных, рассмотрим все функции поддержки DirectDraw. Здесь они несколько расширены для полной поддержки 16-битового оконного режима. Ниже приведено краткое описание каждой функции.

Прототип функции:

```
int DDraw_Init(int width,    // Ширина экрана
        int height,          // Высота экрана
        int bpp,             // Число бит на один пиксель
        int windowed = 0);  // Выбор оконного режима
```

Назначение:

Функция DDraw_Init() запускает и инициализирует подсистему DirectDraw. Вы можете передавать ей любое значение разрешения экрана и глубины цвета, а также выбирать оконный режим, передавая 1 в качестве значения параметра windowed. Если вы выбираете оконный режим, то предварительно должно быть создано окно игры. DDraw_Init() поразному настраивает систему DirectDraw при работе в оконном и полноэкранном режимах. Кроме того, в оконном режиме вы не можете управлять разрешением экрана или глубиной цвета, а потому в этом случае параметр bpp игнорируется.

Эта функция выполняет множество других операций: в 8-битовых режимах она загружает файл paldata2.pal (содержащий палитру по умолчанию), настраивает отсекающий прямоугольник в 8- и 16-битовом режимах во время работы с полным экраном или с окнами, кроме того, в 16-битовом режиме определяет формат пикселей, проверяет, соответствует ли этот формат 5.5.5 или 5.6.5, и на основании этих данных настраивает указатель на функцию генерации цвета пикселей. Это позволяет вызывать функцию RGB16Bit(r,g,b) для создания в 16-битовом режиме RGB-значения цвета, отформатированного надлежащим образом. Разумеется, все значения составляющих цвета не должны выходить за рамки диапазона 0—255. Кроме того, в оконном режиме функция DDraw_Init() полностью освобождает вас от всех проблем, связанных с отсечением.

В случае успешного завершения возвращает TRUE.

Примеры:

```
// Разрешение экрана 800x600, 256 цветов
DDraw_Init(800, 600, 8);

// Оконный режим с размером окна 640x480
// и 16-битовый цвет
DDraw_Init(640, 480, 16, 1);
```

Прототип функции:

```
int DDraw_Shutdown (void);
```

Назначение:

Функция DDraw_Shutdown() закрывает подсистему DirectDraw и освобождает все интерфейсы.

Пример:

```
// Код завершения работы должен содержать вызов
DDraw_Shutdown();
```

Прототип функции:

```
LPDIRECTDRAWCLIPPER
  DDraw_Attach_Clipper(
```

```
    LPDIRECTDRAWSURFACE7  lpdds,
                        // поверхность, к которой присоединяется отсекатель
    int num_rects,      // Количество прямоугольников
    LPRECT clip_list); // Список прямоугольников
```

Назначение:

Функция DDraw_Attach_Clipper() присоединяет отсекатель к передаваемой ей поверхности (в большинстве случаев — ко вторичному буферу). Вы должны также передать в функцию количество прямоугольников, содержащихся в списке отсечения, и указатель на этот список. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Создание области отсечения размером в полный экран
RECT clip_zone = {0,0,SCREEN_WIDTH-1,SCREEN_HEIGHT-1};
DDraw_Attach_Clipper(lpddsback, 1, &clip_zone);
```

Прототип функции:

```
LPDIRECTDRAWSURFACE7 DDraw_Create_Surface(
      int wigth,                      // Ширина поверхности
      int height,                     // Высота поверхности
      int mem_flags,                  // Управляющие флаги
      USHORT color_key_value = 0); // Цветовой ключ
```

Назначение:

Функция DDraw_Create_Surface() используется для создания общей внеэкранной поверхности DirectDraw в системной памяти, видеопамяти или памяти ускоренного графического порта. По умолчанию устанавливается флаг DDSCAPS_OFFSCREENPLAIN. Любые другие управляющие флаги задаются с использованием операции побитового OR и указанного значения по умолчанию. Это стандартные флаги DDSCAP* подсистемы DirectDraw, например DDSCAPS_SYSTEMMEMORY и DDSCAPS_VIDEOMEMORY для системной и оперативной видеопамяти соответственно. Вы можете также задавать значение цветового ключа (сейчас для него установлено значение по умолчанию, равное 0). В случае успешного завершения функция возвращает указатель на новую поверхность, в противном случае — NULL.

Пример:

```
// Создадим поверхность размером 64x64 в видеопамяти
LPDIRECTDRAWSURFACE7 image =
   DDraw_Create_Surface(64, 64, DDSCAPS_VIDEOMEMORY);
```

Прототип функции:

```
int DDraw_Flip(void);
```

Назначение:

Функция DDraw_Flip() выполняет переключение поверхностей, делая вторичную поверхность первичной (при использовании полноэкранных режимов) или копируя внеэкранный вторичный буфер в клиентскую область окна (в оконном режиме). Функция ожидает до тех пор, пока не появится возможность такого переключения, поэтому возврат из функции может произойти не сразу. В случае успешного выполнения возвращает TRUE.

Пример:

```
// Переключение поверхности
DDraw_Flip();
```

Прототип функции:

```
int DDraw_Wait_For_Vsync(void);
```

Назначение:

Функция DDraw_Wait_For_Vsync() ожидает начала очередного периода вертикального обратного хода луча (когда растр достигает нижней границы экрана). В случае успешного завершения функция возвращает TRUE, в противном случае — FALSE.

Пример:

```
// Ожидаем вертикального обратного хода луча
DDraw_Wait_For_Vsync();
```

Прототип функции:

```
int DDraw_Fill_Surface(
  LPDIRECTDRAWSURFACE7 lpdds, // Поверхность
  int          color,         // Цвет
  RECT* client    = NULL );   // Закрашиваемый прямоугольник
```

Назначение:

Функция DDraw_Fill_Surface() используется для закрашивания определенным цветом поверхности или прямоугольника, в некоторой поверхности. Цвет должен иметь формат, установленный для данной поверхности, например задаваться одним байтом, если используется режим 256 цветов, или RGB-дескриптором в высокоцветных режимах. Если вам нужно закрасить всю поверхность, при вызове функции передайте в качестве параметра client значение NULL (которое является значением по умолчанию), в противном случае в качестве этого параметра можно передать указатель на прямоугольник и закрасить только некоторую область поверхности. При успешном завершении функция возвращает TRUE.

Пример:

```
// закрашиваем первичную поверхность цветом 0
DDraw_Fill_Surface (lpddsprimary, 0);
```

Прототип функции:

```
UCHAR* DDraw_Lock_Surface(LPDIRECTDRAWSURFACE7 lpdds,
              int *lpitch);
```

Назначение:

Функция DDraw_Lock_Surface() блокирует переданную ей поверхность (если это возможно) и возвращает указатель типа UCHAR* на эту поверхность, а также обновляет переданную в функцию переменную lpitch, присваивая ей значение шага памяти поверхности. После того как поверхность заблокирована, вы можете использовать ее по своему усмотрению и записывать в нее пиксели, но блиттер будет блокирован, поэтому помните, что разблокировать такую поверхность нужно как можно скорее. После того как поверхность будет разблокирована, указатель, содержащий ее адрес, и шаг видеопамяти, скорее всего, будут содержать неправильные значения, поэтому после разблокирования они не должны использоваться. В случае успешного завершения функция DDraw_Lock_Surface() возвращает ненулевой адрес памяти поверхности, в противном случае — NULL. Помимо этого следует помнить, что если вы работаете в 16-битовом режиме, то на один пиксель приходится 2 байта, но при этом возвращаемый указатель имеет тип UCHAR*, а значение, записываемое в lpitch, указано в байтах, а не в пикселях.

Пример:

```
// Здесь будет храниться шаг видеопамяти
int lpitch = 0;

// Заблокируем изображение, созданное ранее
UCHAR* memory = DDraw_Lock_Surface(image, &lpitch);
```

Прототип функции:

```
int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE7 lpdds);
```

Назначение:

Функция DDraw_Unlock_Surface() позволяет разблокировать поверхность, заблокированную ранее с помощью функции DDraw_Lock_Surface(). Вам нужно только передать в функцию указатель на нужную поверхность. В случае успешного завершения возвращает TRUE.

Пример:

```
// Разблокируем поверхность изображения
DDraw_UnLock_Surface(image);
```

Прототипы функций:

```
UCHAR* DDraw_Lock_Primary_Surface(void);
UCHAR* DDraw_Lock_Back_Surface(void);
```

Назначение:

С помощью этих двух функций блокируются первичная и вторичная отображаемые поверхности. Однако в большинстве случаев вам понадобится блокировать только *вторичную* поверхность (в системе с двойной буферизацией); однако при необходимости можно заблокировать также и *первичную* поверхность. Кроме того, при вызове функции DDraw_Lock_Primary_Surface() в следующих глобальных переменных будут содержаться правильные данные:

```
extern UCHAR*  primary_buffer; // Первичный видеобуфер
extern int     primary_lpitch;      // Шаг видеопамяти
```

Затем вы можете использовать память поверхности по своему усмотрению, но блиттер будет заблокирован. Кроме того, замечу, что в оконных режимах первичный буфер будет указывать на поверхность всего экрана (а не только вашего окна), тогда как вторичный буфер будет указывать на прямоугольник строго того же размера, что и в клиентской области окна. При вызове функции DDraw_Lock_Back_Surface() блокируется поверхность во вторичном буфере, и следующие глобальные переменные будут содержать правильные значения:

```
extern  UCHAR* back_buffer; // Вторичный буфер
extern  int    back_lpitch;      // Шаг видеопамяти
```

НА ЗАМЕТКУ

Ни в коем случае не изменяйте эти глобальные переменные сами. Они используются для отслеживания изменения состояния функциями блокировки. Если вы их измените, может произойти разрушение подсистемы.

Пример:

```
// Заблокируем первичную поверхность и выведем
// пиксель в верхнем левом углу
DDraw_Lock_Primary_Surface();

primary-buffer[0] = 100;
```

Прототип функции:

```
int DDraw_Unlock_Primary_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

Назначение:

Эти функции используются для разблокировки поверхностей, содержащихся в первичном или вторичном буфере. При попытке разблокировать незаблокированную поверхность ничего не произойдет. В случае успешного завершения функции возвращают TRUE.

Пример:

```
// Разблокируем поверхность во вторичном буфере
DDraw_Unlock_Back_Surface();
```

Функции для работы с многоугольниками

Описываемые ниже функции образуют подсистему, предназначенную для работы с двухмерными многоугольниками. Они отнюдь не являются самыми совершенными, быстрыми или современными, а просто решают ваши текущие задачи. В мире существуют и более хорошие способы решения подобных проблем, но вас ведь заинтересовала именно эта книга, не правда ли? Кроме того, некоторые из этих функций имеют две версии: 8- и 16-битовую; для 16-битовых версий характерно наличие дополнительных цифр “16” в конце имени функции.

Прототипы функций:

```
void Draw_Triangle_2D(int x1, int y1,  // Вершины треугольника
              int x2, int y2,
              int x3, int y3,
              int color,                    // Индекс цвета
                                            // в 8-битовом режиме
              UCHAR* dest_buffer,           // Целевой буфер
              int    mempitch);             // Шаг видеопамяти

// 16-битовая версия
void Draw_Triangl_2D16(int x1, int y1,  // Вершины треугольника
               int x2, int y2,
               int x3, int y3,
               int color,                   // RGB-дескриптор цвета
                                            // в 16-битовом режиме
               UCHAR* dest_buffer,          // Целевой буфер
               int mempitch);               // Шаг видеопамяти

// Быстрая версия с фиксированной точкой;
// немного менее точная
void Draw_TriangleFP_2D(int x1, int y1,
               int x2, int y2,
               int x3, int y3,
               int color,
               UCHAR*  dest_buffer,
               int     mempitch);
```

Назначение:

Функции Draw_Triangle_2D* рисуют треугольник в переданном функции буфере памяти, закрашивая его указанным цветом. Треугольник может быть отсечен в соответствии с

текущей областью отсечения, заданной в глобальных переменных, но *не* в отсекателе DirectDraw. Это объясняется тем, что при вычерчивании линий данная функция использует программные средства, а не блиттер. Замечу, что функция Draw_TriangleFP_2D делает то же самое, но для внутренних вычислений использует математические операции в формате с фиксированной точкой; она выполняется немного быстрее и несколько менее точна. Обе функции не возвращают никаких значений.

Пример:

```
// Рисуем треугольник (100,10),(150,50),(50,60) цветом
// с индексом 50 на поверхности во вторичном буфере
Draw_Triangle_2D(100, 10, 150, 50, 50, 60,
           50,          // Цвет с индексом 50
           back_buffer,
           back_lpitch);

// Аналогичный пример для 16-битового режима.
// Рисуем треугольник (100,10),(150,50),(50,60)
// RGB-цветом (255,0,0) на поверхности во вторичном буфере
Draw_Triangle_2D16(100, 10, 150, 50, 50, 60,
            RGB16Bit(255,0,0),
            back_buffer,
            back_lpitch);
```

Прототип функции:

```
inline void Draw_QuadFP_2D(
        int x0, int y0,   // Вершины
        int x1, int y1,
        int x2, int y2,
        int x3, int y3,
        int color,                  // Индекс цвета в 8-битовом режиме
        UCHAR* dest_buffer,   // Целевой видеобуфер
        int    mempitch );        // Шаг видеопамяти
```

Назначение:

Функция Draw_QuadFP_2D() вычерчивает переданный ей четырехугольник, состоящий из двух треугольников. Эта функция не имеет возвращаемого значения.

Пример:

```
// Рисуем четырехугольник, вершины которого должны быть
// упорядочены либо по, либо против часовой стрелки
Draw_QuadFP_2D (0,0, 10,0, 15,20, 5,25,
           100,
           back_buffer, back_lpitch);
```

Прототипы функций:

```
void Draw_Filled_Polygon2D
  (POLYGON2D_PTR poly,     // Вычерчиваемый многоугольник
   UCHAR*       vbuffer,        // Видеобуфер
   int         mempitch);       // Шаг видеопамяти

// 16-битовая версия
void Draw_Filled_Polygon2D16
  (POLYGON2D_PTR poly,    // Вычерчиваемый многоугольник
```

```
  UCHAR*      vbuffer,  // Видеобуфер
  int         mempitch); // Шаг видеопамяти
```

Назначение:

Функции Draw_Filled_Polygon2D*() вычерчивают произвольный заполненный многоугольник, имеющий *n* сторон, и принимают в качестве параметров только вычерчиваемый многоугольник, указатель на видеобуфер и шаг видеопамяти. Хотя при работе как в 8-битовом, так и в 16-битовом режиме функции передаются одни и те же параметры, с точки зрения внутренней реализации эти функции используют разные растеризаторы, поэтому вы должны вызывать именно ту функцию, которая нужна в конкретном случае. Замечу, что эта функция рисует многоугольник относительно его точки отсчета (x_0,y_0), поэтому убедитесь, что они корректно инициализированы. Функция не возвращает никакого значения.

Пример:

```
// Рисуем многоугольник в первичном буфере
Draw_Filled_Polygon2D(&poly,
                      primary_buffer,
                      primary_lpitch);
```

Прототип функции:

```
int Translate_Polygon2D
  (POLYGON2D_PTR poly,  // Переносимый многоугольник
   int dx, int dy);     // Коэффициенты переноса
```

Назначение:

Функция Translate_Polygon2D() переносит точку отсчета (x_0,y_0) заданного многоугольника. Заметьте, что эта функция не преобразует и не видоизменяет реальных вершин, образующих многоугольник. В случае успешного завершения возвращает TRUE.

Пример:

```
// Перенесем многоугольник на 10 пикселей
// по оси X и -5 по оси Y
Translate_Polygon2D (&poly, 10, -5);
```

Прототип функции:

```
int Rotate_Polygon2D
  (POLYGON2D_PTR poly,  // Поворачиваемый многоугольник
   int        theta);   // Значения угла: от 0 до 359
```

Назначение:

Функция Rotate_Polygon2D() поворачивает переданный многоугольник относительно его точки отсчета в направлении, обратном часовой стрелке. Угол должен задаваться целочисленным значением в диапазоне от 0 до 359. В случае успешного завершения возвращает TRUE.

Пример:

```
// Повернем многоугольник на 10 градусов
Rotate_Polygon2D(&poly,10);
```

Прототип функции:

```
int Scale_Polygon2D
  (POLYGON2D_PTR poly,  // Масштабируемый многоугольник
   float sx, float sy );// Коэффициенты масштабирования
```

Назначение:

Функция Scale_Polygon2D() масштабирует переданный многоугольник в соответствии с множителя масштабирования s_x и s_y, которые умножаются на значения координат для осей X и Y соответственно. Не возвращает никакого значения.

Пример:

```
// Равномерно увеличим многоугольник в 2 раза
Scale_Polygon2D(&poly, 2, 2);
```

Двухмерные графические примитивы

Этот набор функций содержит всего понемногу — словом, попурри из графических примитивов. Здесь не должно быть ничего незнакомого для вас, по крайней мере, я так думаю. Некоторые из представленных в этом разделе функций тоже имеют две версии: 8-битовую и 16-битовую; последняя, как обычно, обозначена цифрами 16, указанными в конце ее имени.

Прототип функции:

```
int Draw_Clip_Line(int x0, int y0,  // Начальная точка
            int x1, int y1,         // Конечная точка
            int    color,           // 8-битовый цвет
            UCHAR* dest_buffer,     // Видеобуфер
            int lpitch );           // Шаг видеопамяти

// 16-битовая версия
int Draw_Clip_Line16
      (int x0, int y0,          // Начальная точка
       int x1, int y1,          // Конечная точка
       int color,               // 16-битовый RGB-цвет
       UCHAR* dest_buffer,      // Видеобуфер
       int lpitch );            // Шаг видеопамяти
```

Назначение:

Функция Draw_Clip_Line*() отсекает переданную линию в соответствии с текущим отсекающим прямоугольником, а затем вычерчивает ее в заданном буфере (либо в 8-, либо в 16-битовом режиме). В случае успешного завершения эта функция возвращает TRUE.

Пример:

```
// Нарисуем во вторичном буфере линию, начинающуюся в точке
// (10,10) и заканчивающуюся в точке (100,200)
Draw_Clip_Line(10, 10, 100, 200,
          5,            // Цвет 5
          back_buffer,
          back_lpitch );
```

Прототип функции:

```
int Clip_Line(int &x1, int &y1,  // Начальная точка
         int &x2, int &y2);      // Конечная точка
```

Назначение:

Функция Clip_Line() предназначена главным образом для внутренних целей, но вы также можете вызвать ее для отсечения переданной линии в соответствии с текущим отсекающим прямоугольником. Замечу, что функция изменяет переданные ей в качестве параметров конечные точки, поэтому следует сохранить их, если вы не хотите их поте-

рять. Необходимо отметить, что эта функция ничего не рисует: она только отсекает конечные точки. В случае успешного завершения она возвращает TRUE.

Пример:

```
// Отсекаем линию, которая задана конечными точками
// x1,y1 и x2,y2
Clip_Line(x1, y1, x2, y2);
```

Прототип функции:

```
int Draw_Line(int x0, int y0,  // Начальная точка
        int x1, int y1,        // Конечная точка
        int color,             // 8-битовый индекс цвета
        UCHAR* vb_start,       // Видеобуфер
        int lpitch );          // Шаг видеопамяти
// 16-битовая версия
int Draw_Line16(int x0, int y0,  // Начальная точка
         int x1, int y1,         // Конечная точка
         int color,              // 16-битовый RGB-цвет
         UCHAR* vb_start,        // Видеобуфер
         int lpitch );           // Шаг видеопамяти
```

Назначение:

Функция Draw_Line*() рисует линию в 8- или 16-битовом режиме без какого-либо отсечения, поэтому, перед тем как ее вызвать, убедитесь, что конечные точки линии не выходят за рамки поверхности экрана. Эта функция работает немного быстрее, чем версия с отсечением. В случае успешного завершения возвращает TRUE.

Пример:

```
// Нарисуем во вторичном буфере линию, имеющую конечные
// точки (10, 10) и (100, 200)
Draw_Line16(10, 10, 100, 200,
       RGB16Bit (0, 255, 0),  // Ярко-зеленый
       back_buffer,
       back_lpitch );
```

Прототип функции:

```
inline Draw_Pixel
    (int x, int y,             // Позиция пикселя
     int color,                // Индекс 8-битового цвета
     UCHAR* video_buffer,      // Видеобуфер
     int lpitch );             // Шаг видеопамяти

// 16-битовая версия
inline Draw_Pixel16
    (int x, int y,             // Координаты пикселя
     int color,                // 16-битовый RGB-цвет
     UCHAR* video_buffer,      // Видеобуфер
     int lpitch );             // Шаг видеопамяти
```

Назначение:

Функция Draw_Pixel*() рисует один пиксель в памяти поверхности экрана. В большинстве случае вам не придется создавать объекты на основе пикселей, поскольку на сам этот вызов затрачивается больше времени, чем на непосредственное нанесение пикселя.

Но, если вас не заботят проблемы скорости, можете вызвать эту функцию, и она сделает свое дело. В случае успешного завершения она возвращает TRUE.

Пример:

```
// Нарисуем пиксель в центре экрана, имеющего размеры 640x480
// (в 8-битовом режиме)
Draw_Pixel (320, 240, 100, back_buffer, back_lpitch );
```

Прототип функции:

```
int Draw_Rectangle
    (int x1, int y1,   // Верхний левый угол
     int x2, int y2,   // Нижний правый угол
     int color,        // Дескриптор цвета: индекс для
                       // 8-битовых режимов и RGB-значение
                       // для 16-битовых режимов
     LPDIRECRDRAWSURFACE7 lpdds ); // поверхность  DirectDraw
```

Назначение:

Функция Draw_Rectangle() рисует прямоугольник на переданной поверхности DirectDraw. Эта функция одинаково работает в 8- и 16-битовом режиме, поскольку это чистый DirectDraw-вызов. Замечу, что для выполнения этой функции необходимо, чтобы поверхность не была заблокирована. Кроме того, эта функция использует блиттер, поэтому работает очень быстро. В случае успешного завершения возвращает TRUE.

Пример:

```
// Заполняем экран при помощи блиттера
Draw_Rectangle(0, 0, 639, 479, 0, lpddsback);
```

Прототипы функций:

```
void Hline
    (int x1, int x2,      // Координаты x начальной и
                          // конечной точек
     int y,               // Рисуемая строка
     int color,           // Индекс 8-битового цвета
     UCHAR* vbuffer,      // Видеобуфер
     int lpitch );        // Шаг видеопамяти

// 16-битовая версия
void Hline16
    (int x1, int x2,      // Координата x начальной и
                          // конечной точек
     int y,               // Рисуемая строка
     int color,           // 16-битовый RGB-цвет
     UCHAR*  vbuffer,     // Видеобуфер
     int lpitch );        // Шаг видеопамяти
```

Назначение:

Функции HLine*() рисуют горизонтальную линию, причем очень быстро по сравнению с функцией вычерчивания произвольной линии. Работают как в 8-, так и в 16-битовом режиме. Возвращаемого значения не имеют.

Пример:

```
// Быстро рисуем линию, начинающуюся в точке (10,100)
// и заканчивающуюся в точке (100,100) (в 8-битовом режиме)
```

```
HLine(10, 100, 100,
    20, back_buffer,
    back_lpitch);
```

Прототип функции:

```
void Vline
    (int y1, int y2,      // Координата y начальной и
                          // конечной точек
     int x,               // Рисуемый столбец
     int color,           // Индекс 8-битового цвета
     UCHAR* vbuffer, // Видеобуфер
     int lpitch);         // Шаг видеопамяти

// 16-битовая версия
void Vline16
    ( int y1, int y2,     // Координата y начальной и
                          // конечной точек
     int x,               // Рисуемый столбец
     int color,           // 16-битовый RGB-цвет
     UCHAR*  vbuffer, // Видеобуфер
     int lpitch );        // Шаг видеопамяти
```

Назначение:

Функция VLine*() быстро рисует вертикальную линию. Они работает не так быстро, как HLine*(), но быстрее, чем Draw_Line*(). Поэтому, если вам известно, что линия всегда будет вертикальной, лучше использовать именно функции VLine*(). Функции не возвращают никаких значений.

Пример:

```
// Нарисуем линию, которая начинается в точке (320,0) и
// заканчивается в точке (320,479), используя для этой
// цели 16-битовую версию функции
VLine16(0, 479, 320, RGB16Bit(255, 255, 255),
     primary_buffer,
     primary_lpitch);
```

Прототип функции:

```
void Screen_Transitions
   (int effect,     // Изменение экрана
    UCHAR*  vbuffer, // Видеобуфер
    int lpitch);    // Шаг видеопамяти
```

Назначение:

Функция Screen_Transition() выполняет различные изменения экрана в памяти в соответствии с предварительной заголовочной информацией. Эти изменения деструктивны, поэтому следует сохранять изображение и/или палитру, если они могут потребоваться вам для дальнейшей работы. Изменения с участием цвета работают только в 8-битовых режимах, применяющих палитру; прочие операции работают в любом режиме. Эта функция не возвращает никакого значения.

Пример:

```
// Выполним постепенную заливку первичного экрана черным
// цветом. Функция работает только в 8-битовых режимах
Screen_Transition(SCREEN_DARKNESS, NULL, 0);
```

Прототипы функций:

```
int Draw_Text_GDI
   (char* text,                        // Строка с завершающим нулем
    int x, int y,                      // Позиция
    COLORREF color,                    // Любой RGB-цвет
    LPDIRECTDRAWSURFACE7 lpdds ); // Поверхность
                                       // DirectDraw

int Draw_Text_GDI
   (char* text,                        // Строка с завершающим нулем
    int x, int y,                      // Позиция
    int color,                         // Индекс 8-битового цвета
    LPDIRECTDRAWSURFACE7 lpdds );  // Поверхность
                                       // DirectDraw
```

Назначение:

Функция Draw_Text_GDI() отображает GDI-текст на передаваемой ей поверхности, используя для этого заданные цвет и позицию. Функция перегружена и принимает в качестве параметра цвета либо значение типа COLORREF (с использованием макроса Windows RGB()), либо индекс 8-битового цвета для режимов, которые используют 256 цветов. Заметим, что для нормальной работы функции нужная поверхность не должна быть заблокирована, поскольку данная функция сама блокирует ее для выполнения блиттирования текста с помощью GDI. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Выведем текст RGB-цветом (100,100,0);
// этот вызов должен работать в любом режиме:
// как 8-, так и 16-битовом
Draw_Text_GDI("This is a text", 100, 50,
        RGB (100, 100, 0), lpddsprimary);

// Выведем текст цветом, имеющим индекс 33;
// этот вызов будет работать ТОЛЬКО в 8-битовых режимах
Draw_Text_GDI("This is a text", 100, 50,
        33, lpddsprimary);
```

Математические функции и функции ошибок

До сих пор вам не приходилось вплотную заниматься математикой, поэтому здесь дается лишь немного облегченной информации, ну а серьезная работа с математикой еще ждет вас...

Прототип функции:

```
int Fast_Distance_2D(int x, int y);
```

Назначение:

Функция Fast_Distance() вычисляет расстояние от точки $(0,0)$ до точки (x,y), используя метод быстрой аппроксимации. Эта функция возвращает расстояние, вычисленное с погрешностью, не превышающей 3,5% , отбрасывая при этом дробную часть числа.

Пример:

```
int x1=100, y1=200;  // Первый объект
int x2=400, y2=150;  // Второй объект
```

```
// Вычисляем расстояние между первым и вторым объектами
int dist = Fast_Distance_2D (x1-x2, y1-y2);
```

Прототип функции:

```
float Fast_Distance_3D(float x, float y, float z);
```

Назначение:

Функция Fast_Distance_3D() вычисляет расстояние от точки $(0,0,0)$ до точки (x,y,z), используя метод быстрой аппроксимации. Функция возвращает расстояние, причем погрешность результата не превышает 11%.

Пример:

```
// Вычислим расстояние от точки (0,0,0)
// до точки (100,200,300)
float dist = Fast_Distance_3D(100,200, 300);
```

Прототип функции:

```
int Find_Bounding_Box_Poly2D
  (POLYGON2D_PTR poly,          // Многоугольник
   float &min_x, float &max_x,  // Ограничивающий
   float &min_y, float &max_y); // прямоугольник
```

Назначение:

Функция Find_Bounding_Box_Poly2D() вычисляет наименьший прямоугольник, в котором может содержаться некоторый многоугольник, переданный в качестве параметра poly. В случае успешного завершения функция возвращает TRUE. Обратите внимание, что эта функция использует передачу параметров по ссылке.

Пример:

```
POLYGON2D poly;                    // Предположим, что он уже инициализирован
int min_x, max_x, min_y, max_y;  // В этих переменных
                                   // будет храниться результат

// Находим ограничивающий прямоугольник
Find_Bounding_Box_Poly2D(&poly, min_x, max_x, min_y, max_y)
```

Прототип функции:

```
int Open_Error_File(char* filename);
```

Назначение:

Функция Open_Error_File() открывает на диске файл, в который функция Write_Error() записывает для вас все сообщения об ошибках. В случае успешного завершения функция Open_Error_File() возвращает TRUE.

Пример:

```
// Откроем файл регистрации ошибок
Open_Error_File("errors.log");
```

Прототип функции:

```
int Close_Error_File(void);
```

Назначение:

Функция Close_Error_File() закрывает файл регистрации ошибок, открытый ранее. Если при вызове этой функции оказывается, что файл ошибок уже закрыт, ничего не происходит. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Закроем файл регистрации ошибок
Close_Error_File();
```

Прототип функции:

```
int Write_Error(char* string, ...);
  // строка, определяющая формат сообщения об ошибке
```

Назначение:

Функция Write_Error() записывает сообщение об ошибке в файл регистрации ошибок, открытый ранее. Если этот файл не был открыт, функция просто возвращает FALSE и ничего страшного не происходит. Замечу, что функция принимает переменное количество параметров и использовать ее можно точно так же, как функцию printf(). В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Выводим некоторые данные
Write_Error("\nSystem Starting...");
Write_Error("\nx-vel= %d, y-vel = %d", xvel, yvel);
```

Функции для работы с растровыми изображениями

Очередной набор функций представляет собой подпрограммы, работающие со структурами BIT_IMAGE (описывающими растровые изображения) и BIT_FILE (описывающими файлы с растровыми изображениями). Эти функции предназначены для загрузки 8-, 16-, 24- и 32-битовых растровых изображений, а также для выделения фрагментов из указанных растровых изображений и создания простых объектов типа BITMAP_IMAGE (которые не являются поверхностями DirectDraw). Кроме того, они позволяют выводить изображения в 8- и 16-битовом режиме, но отсечение при этом не поддерживается (это значит, что, если вам понадобится отсечение, придется самостоятельно изменять исходный код соответствующих функций). Как обычно, некоторые функции реализованы в двух версиях: 8- и 16-битовой. Имена файлов, содержащих 16-битовые версии, как всегда заканчиваются цифрами 16.

Прототип функции:

```
int Load_Bitmap_File
  (BITMAP_FILE_PTR bitmap,  // Указатель на структуру
                            // данных о файле с растровым изображением
   char* filename);         // Загружаемый BMP-файл
```

Назначение:

Функция Load_Bitmap_File() загружает с диска .BMP-файл, содержащий растровое изображение, в переданную в качестве параметра структуру BITMAP_FILE, с которой вы сможете работать в дальнейшем. Эта функция загружает 8-, 16- и 24-битовые растровые изображения, а также палитры для 8-битовых .BMP-файлов. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Загружаем с диска файл "andre.bmp"
BITMAP_FILE bitmap_file;

Load_Bitmap_File(&bitmap_file, "andre.bmp");
```

Прототип функции:

```
int Unload_Bitmap_File
  (BITMAP_FILE_PTR bitmap);  // Растровое изображение,
                             // подлежащее закрытию и выгрузке
```

Назначение:

Функция Unload_Bitmap_File() освобождает память, выделенную для буфера изображения, содержащего данные загруженного .BMP-файла — структуру BITMAP_FILE. Эта функция вызывается, когда вы выполнили побитовое копирование изображения и/или закончили работу с некоторым растровым изображением. Структура может быть использована повторно, но вначале вы должны освободить занятую память. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Закрываем только что открытый файл
Unload_Bitmap_File(&bitmap_file);
```

Прототип функции:

```
int Create_Bitmap
  (BITMAP_IMAGE_PTR  image, // Растровое изображение
   int x, int y,            // Начальная позиция
   int width, int height,   // Размеры
   int bpp = 8);            // Число бит на один пиксель:
                            // может быть либо 8, либо 16
```

Назначение:

Функция Create_Bitmap() создает 8- или 16-битовое изображение, содержащееся в системной памяти, имеющее заданные размеры и располагающееся в заданной позиции. Первоначально изображение является пустым и хранит в структуре BITMAP_IMAGE. Растровое изображение не является поверхностью DirectDraw, поэтому здесь не поддерживается ни аппаратное ускорение, ни отсечение. В случае успешного завершения функция возвращает TRUE.

НА ЗАМЕТКУ

Между структурами BITMAP_FILE и BITMAP_IMAGE существует большая разница. Структура BITMAP_FILE описывает .BMP-файл, находящийся на диске, тогда как BITMAP_IMAGE описывает объект, находящийся в системной памяти; этот объект похож на спрайт, который можно перемещать и рисовать.

Пример:

```
// Создадим 8-битовое растровое изображение
// размером 64x64 в позиции(0,0)
BITMAP_IMAGE ship;

Create_Bitmap(&ship, 0, 0, 64, 64, 8);

// Аналогичный пример для работы в 16-битовом режиме
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0, 0, 64, 64, 16);
```

Прототип функции:

```
int Destroy_Bitmap
  (BITMAP_IMAGE_PTR image);  // уничтожаемое растровое
                             // изображение
```

Назначение:

Функция Destroy_Bitmap() используется для освобождения памяти, выделенной в процессе создания объекта типа BITMAP_IMAGE. Вы должны вызвать эту функцию для своего объекта, после того как завершили работу с ним (как правило, во время закрытия игры) или же если данный объект был уничтожен в ходе "кровавой битвы". В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Уничтожаем созданный ранее объект BITMAP_IMAGE
Destroy_Bitmap(&ship);
```

Прототип функции:

```
int Load_Image_Bitmap
  (BITMAP_IMAGE_PTR image,  // Структура для хранения
                            // растрового изображения
   BITMAP_FILE_PTR bitmap,  // Объект файла с растровым
                            // изображением, откуда
                            // выполняется загрузка
   int cx, int cy,          // Координаты сканирования
   int mode);               // Режим считывания изображения

// 16-битовая версия
int Load_Image_Bitmap16
  (BITMAP_IMAGE_PTR image,  // Структура для хранения
                            // растрового изображения
   BITMAP_FILE_PTR bitmap,  // Объект файла с растровым
                            // изображением, откуда
                            // выполняется загрузка
   int cx, int cy,          // Координаты сканирования
   int mode);               // Режим считывания изображения

#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS  1
```

Назначение:

Функция Load_Image_Bitmap() используется для считывания изображения из загруженного ранее объекта типа BITMAP_FILE в передаваемую в качестве параметра структуру BITMAP_IMAGE. Чтобы иметь возможность использовать эту функцию, вначале необходимо загрузить объект типа BITMAPE_FILE и создать структуру BITMAP_IMAGE, а затем вызвать эту функцию для считывания из структуры BITMAPE_FILE изображения того же размера, что и в BITMAP_IMAGE. Функция может работать в двух режимах: ячеек и абсолютных значений.

- В режиме ячеек (BITMAP_EXTRACT_MODE_CELL) изображение считывается в предположении, что все изображения находятся в .BMP-файле в шаблоне; этот шаблон имеет некоторый определенный размер ($m \times n$), а ширина границы между ячейками равна одному пикселю. Как правило, ячейки имеют размеры 8×8, 16×16, 32×32, 64×64 и т.д. Советую обратиться к файлам TEMPLATE*.BMP, находящимся на компакт-диске: они содержат несколько подходящих шаблонов. Ячейки нумеруются слева направо и сверху вниз, а начальная ячейка имеет координаты (0,0).
- В режиме использования абсолютных значений координат (BITMAP_EXTRACT_MODE_ABS) изображение считывается с применением точных значений координат, передаваемых в качестве параметров cx и cy. Этот метод удобно использовать в том случае, если при

загрузке иллюстраций вы хотите использовать изображения различных размеров и единственный .BMP-файл.

Кроме того, вы должны использовать именно ту версию функции, которая подходит для текущего режима. Поэтому обязательно используйте функцию Load_Image_Bitmap() для 8-битовых растровых изображений и функцию Load_Image_Bitmap16() — для 16-битовых изображений.

Пример:

```
// Считаем, что исходный .BMP-файл содержит растровое
// изображение размером 640x480 и имеет 8x8 ячеек
// размером 32x32. Тогда для загрузки в 8-битовом режиме
// 3-й ячейки слева на 2-й строке (ячейки 2,1) вы должны
// сделать следующее

// Загружаем содержимое .BMP-файла в память
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "images.bmp");

// Инициализируем растровое изображение
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0, 0, 32, 32, 8);

// Считываем данные
Load_Image_Bitmap(&ship, &bitmap_file, 2, 1,
          BITMAP_EXTRACT_MODE_CELL);

// Аналогичный пример для работы в 16-битовом режиме

// Загружаем содержимое .BMP-файла в память
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "images24bit.bmp");

// Инициализируем растровое изображение
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0, 0, 32, 32, 16);

// Считываем данные
Load_Image_Bitmap16(&ship, &bitmap_file, 2, 1,
          BITMAP_EXTRACT_MODE_CELL);
```

Для загрузки точно такого же изображения (полагая, что оно находится в том же шаблоне) при использовании абсолютного режима вы должны вычислить соответствующие значения координат. Не забывайте, что граница, разделяющая изображения, имеет ширину в 1 пиксель.

```
Load_Image_Bitmap16(&ship, &bitmap_file,
          2*(32+1)+1, 1*(32+1)+1,
          BITMAP_EXTRACT_MODE_ABS);
```

Прототип функции:

```
int Draw_Bitmap
  (BITMAP_IMAGE_PTR source_bitmap,  // Рисуемое растровое
                                    // изображение
```

```
   UCHAR* dest_buffer,   // Видеобуфер
   int lpitch,           // Шаг видеопамяти
   int transparent);     // Применяется прозрачность?

// 16-битовая версия
int Draw_Bitmap16
   (BITMAP_IMAGE_PTR source_bitmap,  // Рисуемое растровое изображение
    UCHAR* dest_buffer,              // Видеобуфер
    int lpitch,                      // Шаг видеопамяти
    int transparent);                // Применяется прозрачность?
```

Назначение:

Функции Draw_Bitmap*() рисуют переданное им растровое изображение на соответствующей поверхности, находящейся в памяти, возможно, с применением режима прозрачности. Если значение, передаваемое в качестве параметра transparent, равно 1, прозрачность разрешена и любой пиксель с индексом цвета, равным 0, не будет скопирован. Если вы работаете в условиях 16-битовых режимов и изображений, просто воспользуйтесь 16-битовой версией этой функции. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Нарисуем наш маленький кораблик на поверхности,
// находящейся во вторичном буфере (8-битовый режим)
Draw_Bitmap(&ship, back_buffer, back_lpitch, 1);
```

Прототип функции:

```
int Flip_Bitmap
   (UCHAR* image,      // Изображение, переворачиваемое по вертикали
    int bytes_per_line, // Число байт на одну строку
    int height);        // Полное число строк (высота)
```

Назначение:

Обычно функция Flip_Bitmap() служит для внутреннего использования и переворачивает изображение .BMP-файла вверх ногами во время загрузки, для того чтобы оно оказалось правильно расположенным (впрочем, ее можно использовать и для самостоятельного переворота изображения). Эта функция работает при любом значении битовой глубины, поскольку использует такую характеристику, как число байт на одну строку. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Чтобы перевернуть изображение нашего
// кораблика, делаем следующий вызов
Flip_Bitmap(ship->buffer, ship->width, ship_height);
```

Прототип функции:

```
int Scroll_Bitmap
   (BITMAP_IMAGE_PTR  image, // Прокручиваемое растровое изображение
    int dx,                  // Прокрутка вдоль оси x
    int dy = 0);             // Прокрутка вдоль оси y
```

Назначение:

Функция Scroll_Bitmap() используется для горизонтальной или вертикальной прокрутки растрового изображения. Функция работает как для 8-, так и для 16-битовых растровых изображений; она сама определяет их глубину цвета, поэтому вам нужно только вызвать ее. При вызове нужно передать в качестве параметра указатель на прокручиваемое

растровое изображение, а также величины прокрутки по оси X и Y (в пикселях). Эти значения могут быть как положительными, так и отрицательными: положительные обозначают направление вправо и вниз, а отрицательные — влево и вверх. Кроме того, вы можете производить прокрутку как сразу вдоль обеих осей, так и только по одной из них. Выбор за вами. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Прокрутим изображение на 2 пикселя вправо
Scroll_Bitmap(&image, 2, 0);
```

Прототип функции:

```
int Copy_Bitmap
  (BITMAP_IMAGE_PTR dest_bitmap,     // Целевое растровое изображение
   int dest_x, int dest_y,           // Целевая позиция
   BITMAP_IMAGE_PTR source_bitmap,   // Исходное растровое изображение
   int source_x, int source_y,       // Исходная позиция
   int width, int height);           // Размеры копируемого фрагмента
```

Назначение:

Функция Copy_Bitmap() используется для копирования прямоугольной области исходного растрового изображения в заданное растровое изображение. Эта функция сама считывает значение битовой глубины исходного и целевого изображений, поэтому существует только одна версия, которая работает как в 8-, так и в 16-битовом режиме. Разумеется, растровые изображения должны иметь одинаковую глубину цвета. В качестве параметров этой функции передаются следующие данные: целевое растровое изображение, позиция, куда вы хотите скопировать растровое изображение, а также исходное растровое изображение и координаты того места, откуда вы хотите его скопировать, и наконец, ширина и высота копируемого прямоугольника. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Скопируем прямоугольник размером 100x100 из растрового
// изображения bitmap2 в растровое изображение bitmap1
// (из верхнего левого угла первого изображения в такое
// же место другого изображения)
Copy_Bitmap(&bitmap1, 0, 0,
        &bitmap2, 0, 0,
        100, 100);
```

Функции для работы с палитрой

Следующие функции образуют интерфейс для работы с палитрой, содержащей 256 цветов. Эти функции можно применять только в том случае, если ваш дисплей установлен в режим использования 256 цветов (т.е. цвет имеет 8-битовую глубину). Кроме того, если вы запускаете систему в 8-битовом режиме с помощью функции DDraw_Init(), она будет загружать с диска палитру по умолчанию (или, по крайней мере, попытается это сделать). Файлами палитры, задаваемыми по умолчанию, являются следующие: palette1.pal, palette2.pal и palette3.pal.

Прототип функции:

```
int Set_Palette_Entry
  (int color_index,          // Индекс изменяемого цвета
   PALETTEENTRY color);  // Цвет
```

Назначение:

Функция Set_Palette_Entry() используется для изменения одного цвета в цветовой палитре. Вы просто передаете ей индекс цвета (0–255), а также указатель на структуру PALETTEENTRY, содержащую цвет, после чего следующий кадр будет содержать обновленный цвет. Заметим, что эта функция работает достаточно медленно, поэтому, если вам нужно изменить всю палитру, воспользуйтесь другой функцией — Set_Palette(). В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Делаем цвет 0 черным
PALETTEENTRY black = {0, 0, 0, PC_NOCOLLAPSE};
Set_Palette_Entry(0, &black);
```

Прототип функции:

```
int Get_Palette_Entry
  (int color_index,           // Индекс считываемого цвета
   LPPALETTEENTRY color); // Место для хранения цвета
```

Назначение:

Функция Get_Palette_Entry() считывает элемент текущей палитры. Она работает очень быстро, так как считывает данные из теневой палитры, находящейся в оперативной памяти. Это значит, что вы можете вызывать ее столько раз, сколько захотите: она никак не влияет на работу аппаратных средств. Однако если вы сделали изменения в системной палитре не при помощи функций Set_Palette_Entry() или Set_Palette(), теневая палитра не будет содержать обновленных значений и считываемые данные могут быть ошибочными. В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Считываем сотый элемент палитры
PALETTEENTRY color;
Get_Palette_Entry(100, &color);
```

Прототип функции:

```
int Save_Palette_To_File
  (char* filename,            // Имя файла, в котором будет
                              // сохранена палитра
   LPPALETTEENTRY  color); // Сохраняемая палитра
```

Назначение:

Функция Save_Palette_To_File() сохраняет данные переданной палитры в ASCII-файле на диске для последующего считывания или обработки. Эту функцию очень удобно использовать в том случае, когда вы создаете палитру “на ходу” и хотите сохранить ее на диске. Но эта функция предполагает, что передана палитра, состоящая из 256 элементов, так что будьте очень внимательны! В случае успешного завершения функция возвращает TRUE.

Пример:

```
PALETTEENTRY my_palette[256];
// Считаем, что здесь хранится созданная палитра

// Сохраняем созданную палитру.
// Имя файла может быть любым, но предпочтительно
// использовать расширение .pal
Save_Palette_To_File("/palettes/custom1.pal", my_palette);
```

Прототип функции:

```
int Load_Palette_From_File
  (char* filename,              // Имя файла, откуда загружается палитра
   LPPALETTEENTRY palette);  // Палитра загружается в это место в памяти
```

Назначение:

Функция Load_Palette_From_File() используется для загрузки с диска 256-цветной палитры, сохраненной ранее с помощью функции Save_Palette_To_File(). Вам просто нужно передать в функцию имя файла, а также переменную для сохранения всех 256 элементов палитры, и она будет загружена с диска в эту структуру данных. Эта функция *не* загружает данные в аппаратную палитру; вы должны это делать самостоятельно, используя функцию Set_Palette(). В случае успешного завершения функция возвращает TRUE.

Пример:

```
// Загрузим палитру, сохраненную ранее
PALETTEENTRY disk_palette[256];

Load_Palette_From_File("/palettes/custom1.pal",
                       &disk_palette);
```

Прототип функции:

```
int Set_Palette
  (LPPALETTEENTRY set_palette); // Палитра, загружаемая аппаратно
```

Назначение:

Функция Set_Palette() загружает переданную палитру в аппаратные средства, а также обновляет теневую палитру. В случае успешного завершения возвращает TRUE.

Пример:

```
// Загрузим палитру
Set_Palette(disk_palette);
```

Прототип функции:

```
int Save_Palette
  (LPPALETTEENTRY sav_palette); // Переменная для сохранения палитры
```

Назначение:

Функция Save_Palette() считывает данные аппаратной палитры в переменную sav_palette с тем, чтобы вы могли сохранить их на диске или работать с ними. Переменная sav_palette должна иметь размеры, позволяющие ей сохранять все 256 элементов палитры.

Пример:

```
// Считываем данные текущей аппаратной палитры
PALETTEENTRY hardware_palette[256];
Save_Palette(hardware_palette);
```

Прототип функции:

```
int Rotate_Colors
  (int start_index, // Начальный индекс 0...255
   int end_index);  // Конечный индекс 0...255
```

Назначение:

Функция Rotate_Color() выполняет циклический сдвиг банка цветов в 8-битовых режимах. Она непосредственно работает с аппаратной цветовой палитрой. В случае успешного завершения возвращает TRUE.

Пример:

```
// Выполняем циклический сдвиг всей палитры
Rotate_Colors(0,255);
```

Прототип функции:

```
int Blink_Colors
  (int command,                 // Команда мигания
   BLINKER_PTR new_light,       // Данные
   int id);                     // Идентификатор
```

Назначение:

Функция Blink_Colors() используется для создания асинхронной анимации с использованием палитры. Объяснение этой функции потребовало бы слишком много времени, поэтому советую обратиться к главе 7, "Постигаем секреты DirectDraw и растровой графики", содержащей ее подробное описание.

Служебные функции

Очередной набор состоит из служебных функций, которые мне приходится часто использовать. Я подумал, что они могут быть полезны и для вас.

Прототип функции:

```
DWORD Get_Clock(void);
```

Назначение:

Функция Get_Clock() возвращает текущее время (в миллисекундах) с момента начала работы Windows.

Пример:

```
// Получим текущее значение таймера
DWORD start_time = Get_Clock();
```

Прототип функции:

```
DWORD Start_Clock(void);
```

Назначение:

Функция Start_Clock() вызывает функцию Get_Clock() и сохраняет время в глобальной переменной. Затем вы можете вызвать функцию Wait_Clock(), которая ожидает прошествия определенного времени с момента последнего вызова Start_Clock(). Возвращает показание таймера в момент вызова.

Пример:

```
// Запустим отсчет времени и установим
// значение глобальной переменной
Start_Clock();
```

Прототип функции:

```
DWORD Wait_Clock
  (DWORD count); // Время ожидания в миллисекундах
```

Назначение:

Функция Wait_Clock() осуществляет ожидание в течение определенного количества миллисекунд с момента последнего обращения к функции Start_Clock(). Возвращает текущее значение таймера в момент вызова. Возврат из этой функции не происходит до тех пор, пока с момента последнего вызова функции Start_Clock() не пройдет указанное время.

Пример:

```
// Ожидаем в течение 30 миллисекунд
Start_Clock();

// ...некоторый код...

Wait_Clock(30);
```

Прототип функции:

```
int Collision_Test
   (int x1, int y1,    // Верхний левый угол объекта 1
    int w1, int h1,    // Ширина и высота объекта 1
    int x2, int y2,    // Верхний левый угол объекта 2
    int w2, int h2);  // Ширина и высота объекта 2
```

Назначение:

Функция Collision_Test() проверяет, не перекрываются ли два данных прямоугольника. Вы должны передать в качестве параметров координаты левого верхнего угла каждого прямоугольника, а также их ширину и высоту. Функция возвращает TRUE, если прямоугольники перекрываются, и FALSE — если не перекрываются.

Пример:

```
// Перекрываются ли два прямоугольника

if (Collision_Test
    (ship1->x, ship1->y, ship1->width, ship1->height,
     ship2->x, ship2->y, ship2->width, ship2->height))
{
   // столкновение
} // if
```

Прототип функции:

```
int Color_Scan
   (int x1, int y1,             // Верхний левый угол прямоугольника
    int x2, int y2,             // Нижний правый угол прямоугольника
    UCHAR   scan_start,     // Начальный цвет поиска
    UCHAR   scan_end,       // Конечный цвет поиска
    UCHAR*  scan_buffer,  // Сканируемый буфер
    int     scan_lpitch);      // Шаг видеопамяти

// 16-битовая версия
int Color_Scan16
   (int x1, int y1,             // Верхний левый угол прямоугольника
    int x2, int y2,             // Нижний правый угол прямоугольника
    UCHAR   scan_start,     // Первое RGB-значение поиска
    UCHAR   scan_end,       // Второе RGB-значение поиска
    UCHAR*  scan_buffer,  // Сканируемый буфер
    int     scan_lpitch);     // Шаг видеопамяти
```

Назначение:

Функции Color_Scan*() используют другой алгоритм обнаружения столкновений, который сканирует прямоугольник, для того чтобы найти либо единственное 8-битовое значение, либо последовательность значений, принадлежащих некоторому непрерывно-

му диапазону (в 8-битовых режимах), либо (в случае использования 16-битового режима) обнаружить до двух RGB-значений. Вы можете использовать эту функцию для того, чтобы определить, имеется ли в заданной области тот или иной цвет. Возвращает TRUE, если цвета (или один цвет) были найдены.

Пример:

```
// Ищем цвета в диапазоне от 122 по 124 включительно
// (в 8-битовом режиме)
Color_Scan(10,10,50,50,122,124,back_buffer,back_lpitch);

// Ищем RGB-цвета (10,30,40) и (100,0,12)
Color_Scan(10,10,50,50,
      RGB16Bit(10,30,40),RGB16Bit(100,0,12),
      back_buffer, back_lpitch);
```

Работа с объектами блиттера

Хотя вы можете сделать практически все, что хотите, с помощью типа BITMAP_IMAGE, такой подход имеет один серьезный недостаток — он не использует поверхности DirectDraw и потому не поддерживает аппаратное ускорение. Поэтому я создал новый тип, который очень похож на спрайт, и назвал его *объектом блиттера* (Blitter Object — BOB). Для тех, кто не занимался программированием игр раньше, хочу пояснить, что спрайт — это просто объект, который можно перемещать по экрану, как правило, не нарушая фона. В данном случае это не так, а потому я назвал свой анимационный объект не спрайтом, а объектом блиттера — только и всего.

До настоящего момента вы еще не встречались в книге с каким-либо исходным кодом, использующим объекты блиттера, но вы знаете все, что нужно для создания такого объекта. Ограниченный объем книги не позволяет разобрать весь код, связанный с объектами блиттера, так что вам придется разбираться с ним самостоятельно — он находится в файле T3DLIB1.CPP. Я намерен познакомить вас со всеми функциями для работы с объектами блиттера, входящими в состав этого файла, а дальше вы можете делать с ними все, что хотите: использовать этот код, распечатать его или сжечь в камине. Прежде чем перейти к остальным, не графическим компонентами DirectX, я просто хочу показать вам хороший пример использования поверхностей DirectDraw и возможностей ускорения.

По сути, объект блиттера является графическим объектом, представленным одной или несколькими (вплоть до 64) поверхностями DirectDraw. Вы можете перемещать объект, выводить и анимировать его. Кроме того, объекты блиттера отсекаются текущим отсекателем DirectDraw. На рис. 8.55 показан объект блиттера и его связь с соответствующими кадрами анимации. Создание анимационных последовательностей — очень ценная возможность подсистемы объектов блиттера.

Все функции объекта блиттера возвращают TRUE в случае успешного завершения и FALSE в противном случае. И, разумеется, объекты блиттера, могут работать как в 8-, так и в 16-битовых режимах. Поскольку объекты блиттера в той или иной степени являются объектами DirectDraw, лишь немногие функции нуждаются в 16-битовой версии (это характерно главным образом для функций рисования и загрузки): большинство объектов блиттера достаточно абстрактны, поэтому выбор режима не оказывает на них никакого влияния. В тех случаях, где важен выбор режима, имеются две версии функций — для 8-битовых и для 16-битовых режимов. Последние носят то же название, что и 8-битовые функции, но их имена заканчиваются числом 16.

Приступим к рассмотрению этих функций.

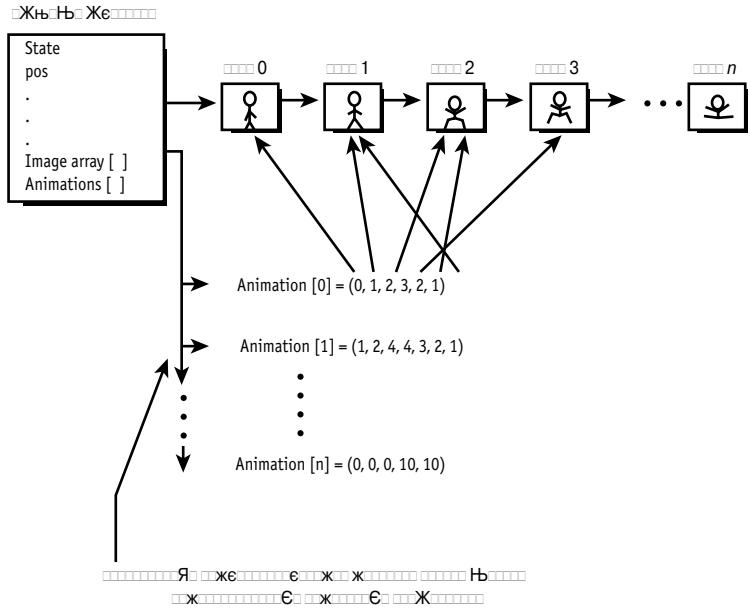


Рис. 8.55. Анимация объекта блиттера

Прототип функции:

```
int Create_BOB
   (BOB_PTR bob,                  // Указатель на создаваемый объект
    int x, int y,                 // Начальная позиция объекта блиттера
    int width, int height,        // Размеры объекта блиттера
    int num_frames,               // Полное число кадров объекта блиттера
    int  attr,                    // Атрибуты объекта блиттера
    int mem_flags,                // Флаги памяти поверхности (0 означает видеопамять)
    USHORT color_key_value=0,     // Значение цветового ключа
    int bpp=8);                   // глубина цвета объекта
```

Назначение:

Функция Create_BOB() создает единственный объект блиттера, предназначенный либо для 8-, либо для 16-битового режима экрана. Помимо создания отдельной поверхности DirectDraw для каждого кадра, эта функция задает все внутренние переменные. Имена почти всех параметров говорят сами за себя; единственный параметр, который требует небольшого пояснения — это атрибут attr. Подробное описание каждого значения атрибута содержится в табл. 8.4. Вы можете объединять их, используя операцию побитового OR, и передавать в качестве значения этого параметра.

Таблица 8.4. Допустимые атрибуты объекта блиттера

Значение	*Описание*
BOB_ATTR_SINGLE_FRAME	Создает объект блиттера, имеющий один кадр
BOB_ATTR_MULTI_FRAME	Создает объект блиттера с несколькими кадрами; анимация объекта блиттера представляет собой линейную последовательность, состоящую из кадров 0...*n*

Значение	*Описание*
BOB_ATTR_MULTI_ANIM	Создает многокадровый объект блиттера, который поддерживает анимационные последовательности
BOB_ATTR_ANIM_ONE_SHOT	При задании этого флага анимационная последовательность воспроизводится только один раз и затем останавливается. В этот момент будет установлена внутренняя переменная anim_state. Чтобы начать повторное воспроизведение анимации, нужно сбросить эту переменную
BOB_ATTR_BOUNCE	Этот флаг вынуждает объект блиттера отскакивать от границ экрана подобно мячику. Он действует только в том случае, если вы используете функцию Move_BOB()
BOB_ATTR_WRAPAROUND	При задании этого флага объект блиттера по мере перемещения за границу с одной стороны экрана появляется с другой стороны. Этот флаг действует только в том случае, если вы используете функцию Move_BOB()

Пример:

Ниже приведено несколько примеров создания объектов блиттера. Вначале создается однокадровый 8-битовый объект блиттера размером 96×64 в точке с координатами (50, 100):

```
BOB  car;  // Объект блиттера
// Создаем объект блиттера
if (!Create_BOB(&car, 50, 100,
        96, 64, 1, BOB_ATTR_SINGLE_FRAME, 0))
  { /* ошибка */ }

// Последние два параметра опущены, поскольку для них
// задаются значения по умолчанию: 0 - для цветового ключа
// и 8 - для bpp
```

Теперь создадим многокадровый 16-битовый объект блиттера размером 32×32 (этот объект блиттера имеет 8 кадров):

```
BOB  ship;  // Объект блиттера

// Создаем объект блиттера
if (!Create_BOB(&ship, 0, 0,
        32, 32, 8, BOB_ATTR_MULTI_FRAME,0, 0,16))
  { /* ошибка */ }
```

И наконец, создадим многокадровый 8-битовый объект блиттера, который поддерживает анимационные последовательности:

```
BOB  greeny;  // Объект блиттера

// Создаем объект блиттера
if (!Create_BOB(&greeny, 0, 0,
        32, 32, 32, BOB_ATTR_MULTI_ANIM, 0, 0,0))
  { /* ошибка */ }
```

ВНИМАНИЕ

При создании объектов блиттера крайне важно задавать нужную цветовую глубину (или число бит на один пиксель). Если вы не сделаете этого, в системе может произойти фатальный сбой. Не забывайте, что по умолчанию используется функция, предназначенная для 8-битовых режимов.

Прототип функции:

```
int Destroy_BOB
  (BOB_PTR bob); // Указатель на уничтожаемый объект
```

Назначение:

Функция Destroy_BOB() уничтожает ранее созданный объект блиттера. Она вызывается, если вы завершили работу с объектом блиттера и хотите освободить занимаемую им память. Указанный вызов справедлив как для 8-, так и для 16-битовых объектов блиттера.

Пример:

```
// Чтобы уничтожить созданный ранее объект
// блиттера, вы должны сделать следующее:
Destroy_BOB(&greeny);
```

Прототип функции:

```
int Draw_BOB
  (BOB_PTR bob,  // Указатель на рисуемый
                 // объект блиттера
   LPDIRECTDRAWSURFACE7 dest); // Поверхность, на которой
                 // будет нарисован объект
                 // блиттера

// 16-битовая версия
int Draw_BOB16
  (BOB_PTR bob,  // Указатель на рисуемый
                 // объект блиттера
   LPDIRECTDRAWSURFACE7 dest); // Поверхность, на которой
                 // будет нарисован объект
                 // блиттера
```

Назначение:

Функции Draw_BOB*() очень мощные. Они выводят переданный им объект блиттера на заданной поверхности DirectDraw. Объект блиттера рисуется в текущей позиции и текущем кадре, который определяется соответствующими параметрами анимации. Стандартная функция Draw_BOB() используется для 8-битовых режимов, а ее 16-битовая версия — Draw_BOB16() — для 16-битовых режимов экрана.

ВНИМАНИЕ

Для работы функции необходимо, чтобы нужная поверхность не была заблокирована.

Пример:

```
// Так вы могли бы разместить 8-битовый
// многокадровый объект блиттера в точке (10,50)
// и нарисовать его первый кадр на вторичной
// поверхности
BOB ship;  // Объект блиттера

// Создаем объект блиттера
if (!Create_BOB(&ship, 0, 0,
```

```
        32, 32, 8, BOB_ATTR_MULTI_FRAME, 0))
  { /* Ошибка */ };

// Задаем положение объекта и кадр
ship.x =50;
ship.y =50;
ship.curr_frame = 0;
// Рисуем объект блиттера
Draw_BOB(&ship, lpddsback);

// Аналогичный пример, использующий 16-битовый
// объект блиттера

BOB ship;  // Объект блиттера

// Создаем объект блиттера
if (!Create_BOB(&ship, 0, 0,
        32, 32, 8, BOB_ATTR_MULTI_FRAME, 0))
  { /* Ошибка */ };

// Задаем положение объекта и кадр
ship.x =50;
ship.y =50;
ship.curr_frame = 0;
// Рисуем объект блиттера
Draw_BOB16(&ship, lpddsback);
```

Прототип функции:

```
int Draw_Scaled_BOB
   (BOB_PTR bob,                    // Указатель на объект
    int swidth, int sheight,        // Новые ширина и высота
                                    // объекта блиттера
    LPDIRECTDRAWSURFACE7 dest); // Поверхность, на которой
                                    // рисуется объект блиттера

// 16-битовая версия
int Draw_Scaled_BOB16
   (BOB_PTR bob,                    // Указатель на объект
    int swidth, int sheight,        // Новые ширина и высота
                                    // объекта блиттера
    LPDIRECTDRAWSURFACE7 dest); // Поверхность, на которой
                                    // рисуется объект блиттера
```

Назначение:

Функции Draw_Scaled_BOB*() работают точно так же, как и функции Draw_BOB*(), с тем отличием, что вы можете передавать любые значения ширины и высоты рисуемого объекта блиттера, а подсистема будет соответственно увеличивать или уменьшать объект. Это замечательная возможность, и если вы можете использовать аппаратное ускорение, то получаете великолепный способ масштабирования объекта и придания ему схожести с пространственными объектами. Разумеется, следует использовать именно ту версию функции, которая соответствует конкретному значению текущей глубины цвета.

Пример:

```
// Пример рисования корабля размером 128x128,
// хотя при создании его размеры составляли
// только 32x32 пикселей
Draw_Scaled_BOB(&ship, 128, 128, lpddsback);
```

Прототип функции:

```
int Load_Frame_BOB
   (BOB_PTR bob,                // Указатель на объект
    BITMAP_FILE_PTR bitmap, // Указатель на файл
    int frame,                  // Номер кадра, в который
                                // помещается изображение
    int cx, int cy,             // Позиция ячейки или
                                // абсолютная позиция
    int mode);                  // Режим считывания - тот же,
                                // что и в Load_Frame_Bitmap()

// 16-битовая версия
int Load_Frame_BOB16
   (BOB_PTR bob,                // Указатель на объект
    BITMAP_FILE_PTR bitmap, // Указатель на файл
    int frame,                  // Номер кадра, в который
                                // помещается изображение
    int cx, int cy,             // Позиция ячейки или
                                // абсолютная позиция
    int mode);                  // Режим считывания - тот же,
                                // что и в Load_Frame_Bitmap()
```

Назначение:

Функции Load_Frame_BOB*() работают точно так же, как и Load_Frame_Bitmap*(), поэтому за подробностями обратитесь к описанию последних. Единственный дополнительный параметр — frame — задает кадр загрузки. Если вы создаете объект блиттера, имеющий четыре кадра, все кадры будут загружаться по очереди. И конечно, следует использовать нужную версию функции, соответствующую текущему значению глубины цвета.

Пример:

```
// Пример загрузки (в режиме ячеек)
// 4 кадров из файла с растровым изображением
// в 8-битовый объект блиттера

BOB ship;  // Объект блиттера
// Загружаем кадры 0,1,2,3 из ячеек (0,0), (1,0),
// (2,0), (3,0) из файла с именем bitmap8bit, содержащего
// растровое изображение

for(int index=0; index<4; index++)
   Load_Frame_BOB(&ship, &bitmap8bit,
            index, index, 0,
            BITMAP_EXTRACT_MODE_CELL);

// Пример загрузки 4 кадров из файла
// с растровым изображением в 16-битовый
// объект блиттера (в режиме ячеек)
```

```
BOB  ship;  // Объект блиттера
// Загружаем кадры 0,1,2,3 из ячеек (0,0), (1,0),
// (2,0), (3,0) из файла под названием bitmap8bit,
// содержащего растровое изображение

for(int index=0; index<4; index++)
   Load_Frame_BOB16(&ship, &bitmap8bit,
              index, index, 0,
              BITMAP_EXTRACT_MODE_CELL);
```

Прототип функции:

```
int Load_Animation_BOB
   (BOB_PTR bob,     // Объект блиттера, в который
                     // загружается анимация
    int  anim_index, // Загружаемая анимация: 1...15
    int  num_frames, // Число кадров анимации
    int* sequence);  // Указатель на массив, содержащий
                     // последовательность анимации
```

Назначение:

Функция Load_Animation_BOB() требует некоторых пояснений. Эта функция используется для загрузки одного из 16 внутренних массивов в объект блиттера, который содержит анимационные последовательности. Каждая последовательность содержит массив индексов или номеров кадров, подлежащих последовательному воспроизведению.

Пример:

Пусть у нас есть объект блиттера, содержащий 8 кадров: 0, 1....7, и есть четыре анимации, определенные следующим образом:

```
int anim_walk[]  = {0,1, 2, 1, 0}
int anim_fire[]  = {5,6,0}
int anim_die[]   = {3,4}
int anim_sleep[] = {0,0,7,0,0}
```

Тогда для загрузки анимации в 16-битовый объект блиттера вы должны сделать следующее:

```
// Создаем объект блиттера, содержащий несколько
// анимационных последовательностей
if (!Create_BOB(&alien, 0, 0, 32, 32, 8,
          BOB_ATTR_MULTI_ANIM, 0, 0, 16))
   { /* ошибка */}

// Загружаем кадры объекта блиттера.
// Загружаем "ходьбу" в анимацию 0
Load_Animation_BOB (&alien, 0, 5, anim_walk);

// Загружаем "огонь" в анимацию 1
Load_Animation_BOB (&alien, 1, 3, anim_fire);

// Загружаем "умирание" в анимацию 2
Load_Animation_BOB (&alien, 2, 2, anim_die);

// Загружаем "сон" в анимацию 3
Load_Animation_BOB (&alien, 3, 5, anim_sleep);
```

После загрузки анимаций вы можете задать активную анимацию и воспроизвести ее с помощью описанных ниже функций.

Прототип функции:

```
int Set_Pos_BOB
  (BOB_PTR bob,   // Указатель на объект блиттера,
                  // для которого задается позиция
   int x, int y);  // Новая позиция объекта блиттера
```

Назначение:

Функция Set_Pos_BOB() — это очень простой способ задания позиции объекта блиттера. Она просто присваивает значения внутренним переменным (x,y), но иметь ее в своем арсенале совсем неплохо.

Пример:

```
// Задаем позицию объекта блиттера
Set_Pos_BOB(&alien, player_x, plaeyr_y);
```

Прототип функции:

```
int Set_Vel_BOB
  (BOB_PTR  bob,   // Указатель на объект блиттера,
                   // для которого задается скорость
   int xv, int yv);  // Новая скорость вдоль осей x, y
```

Назначение:

Каждый объект блиттера обладает внутренней скоростью, ее значения хранятся в переменных xv и yv. Функция Set_Vel_BOB() просто заменяет эти значения новыми величинами, передаваемыми в функцию в качестве параметров. Значения скорости в объекте блиттера не играют никакой роли до тех пор, пока вы не прибегнете к функции Move_BOB() для перемещения объектов блиттера. Но, даже если вы не используете функцию Move_BOB(), вы можете использовать xv и yv для самостоятельного отслеживания скорости объекта блиттера.

Пример:

```
// Заставим объект блиттера перемещаться
// по горизонтальной прямой
Set_Vel_BOB(&alien, 10, 0);
```

Прототип функции:

```
int Set_Anim_Speed_BOB
  (BOB_PTR bob,  // Указатель на объект блиттера
   int speed);      // Скорость анимации
```

Назначение:

Функция Set_Anim_Speed_BOB() задает внутреннюю скорость анимации для объекта блиттера — anim_count_max. Чем больше эта величина, тем медленнее анимация; чем меньше эта величина (минимальная — 0), тем быстрее анимация. Однако эта функция имеет значение только в том случае, если вы используете внутреннюю функцию анимации объекта блиттера Animate_BOB(). Разумеется, до этого должен быть создан объект блиттера, имеющий несколько кадров.

Пример:

```
// зададим частоту кадров 30 fps
Set_Anim_Speed_BOB(&alien, 30);
```

Прототип функции:

```
int Set_Animation_BOB
  (BOB_PTR bob,    // Указатель на объект блиттера,
```

```
                    // для которого задается анимация
  int anim_index); // Индекс задаваемой анимации
```

Назначение:

Функция Set_Animation_BOB() задает текущую анимацию, которая будет воспроизводиться объектом блиттера. В описанном выше примере использования функции Load_Animation_BOB() создано четыре таких анимации.

Пример:

```
// Сделаем активной анимационную последовательность номер 2
Set_Animation_BOB(&alien, 2)
```

НА ЗАМЕТКУ Вызов этой функции также сбрасывает указатель кадров анимационной последовательности объекта блиттера, устанавливая его на первый кадр.

Прототип функции:

```
int Animate_BOB
  (BOB_PTR bob); // Указатель на объект блиттера,
                 // подлежащего анимации
```

Назначение:

Функция Animate_BOB() анимирует объект блиттера. Обычно она вызывается один раз за кадр для обновления анимационной последовательности объекта блиттера.

Пример:

```
// ... все стираем ...
// ... все перемещаем ...
// ... все анимируем ...
Animate_BOB(&alien);
```

Прототип функции:

```
int Move_BOB
  (BOB_PTR bob);  // Указатель на перемещаемый объект
```

Назначение:

Функция Move_BOB()перемещает объект блиттера в соответствии со значениями приращений xv и yv и затем, в зависимости от значений атрибутов, объект блиттера будет либо отскакивать от стенок, либо переходить на другую сторону экрана (либо не делать ничего этого). Подобно функции Animate_BOB(), вы вызываете эту функцию в основном цикле один раз: сразу после (или перед) функцией Animate_BOB().

Пример:

```
// Анимируем объект блиттера
Animate_BOB(&alien);

// Выполняем перемещения
Move_BOB(&alien);
```

Прототип функции:

```
int Hide_BOB
  (BOB_PTR  bob);  // Указатель на скрываемый объект блиттера
```

Назначение:

Функция Hide_BOB() задает флаг объекта блиттера, делающий его невидимым; функция Draw_BOB() при этом не будет выводить данный объект блиттера.

Пример:

```
// Скроем объект блиттера
Hide_BOB(&alien);
```

Прототип функции:

```
int Show_BOB
  (BOB_PTR bob); // указатель на видимый объект блиттера
```

Назначение:

Функция Show_BOB() задает флаг объекта блиттера, делающий его видимым, после чего указанный объект блиттера будет отображаться (отменяет действие функции Hide_BOB()). Ниже приведен пример, когда объект блиттера делается сначала невидимым, а затем видимым (например, из-за того, что вы выводите объект GDI или еще что-нибудь и не хотите, чтобы объект блиттера его закрывал).

Пример:

```
Hide_BOB(&alien);

// Вызываем функции, выводящие объекты блиттера, GDI и т.д.

Show_BOB(&alien);
```

Прототип функции:

```
int Collision_BOBS
  (BOB_PTR bob1,  // Указатель на первый объект блиттера
   BOB_PTR bob2); // Указатель на второй объект блиттера
```

Назначение:

Функция Collision_BOBS() проверяет, не перекрываются ли ограничивающие прямоугольники двух объектов блиттера. Она может быть использована в игре для обнаружения столкновений.

Пример:

```
// Проверяем, не столкнулся ли один объект блиттера
// (игрок) с другим объектом блиттера (ракетой)
if (!Collision_BOBS(&missile, &player))
  {/* генерируем звук взрыва */}
```

Резюме

Вот и подошла к концу эта нескончаемая глава... Просто материал, который хотелось бы рассмотреть, столь обширен, что пришлось познакомить вас только с самыми важными темами.

Считайте эту главу учебником для начинающих сразу по нескольким темам: растеризации, отсечению, черчению линий, матрицам, обнаружению столкновений, синхронизации, прокрутке, изометрическим подсистемам... Вы видите, как велико одно только перечисление затронутых в этой главе тем! Но довольно отступлений! Сейчас, когда вы располагаете библиотекой, предназначенной для работы с растровыми изображениями и многоугольниками, вы можете с пониманием дела работать над исходными текстами демонстрационных программ во время чтения остальных глав книги. Не стесняйтесь вносить в них свои изменения, ибо применение полученных знаний на практике — лучший способ их закрепления.

ГЛАВА 9

DirectInput

Я хорошо помню, как я разрабатывал интерфейс джойстика на TTL-логике для того, чтобы игры на моем Atari 800 могли поддерживать работу четырех игроков через один 9-контактный порт. Могу ли я после этого считать себя человеком пресыщенным? Во всяком случае, с тех времен устройства ввода прошли долгий путь, и теперь их поддержка осуществляется при помощи системы DirectX. В этой главе рассматривается подсистема DirectInput и некоторые общие алгоритмы ввода. Здесь затрагиваются следующие темы.

- Обзор DirectInput
- Клавиатура
- Мышь
- Джойстик
- Объединение вводов
- Библиотека программ ввода

Цикл ввода: повторение пройденного

Сейчас самое время проанализировать общую схему игры и связь между вводом и циклом событий. На рис. 9.1 представлен типичный игровой цикл, состоящий из стирания, перемещения, перерисовывания, ожидания и повторения — словом, здесь есть все, что нужно для видеоигры. Конечно, это весьма упрощенное представление. В среде Win32/DirectX придется добавить изрядное количество кода, предназначенного для настройки приложения, его завершения, а также обработки событий операционной системы Windows с тем, чтобы получить работающее приложение Windows/DirectX. Но если вы один раз позаботитесь обо всем этом, у вас будет практически все необходимое.

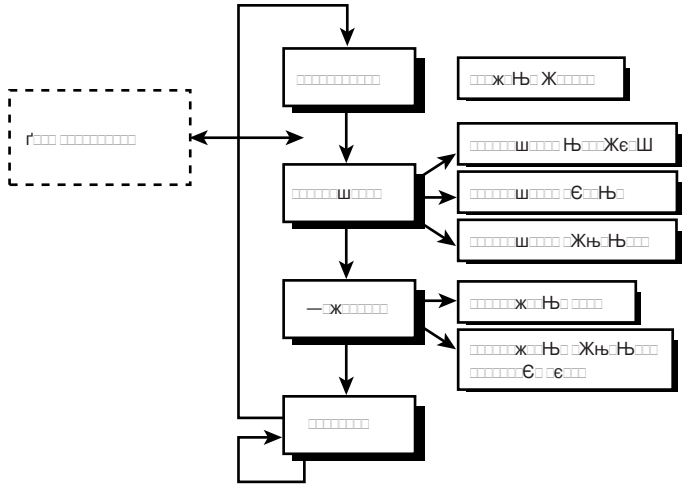


Рис. 9.1. Типичный цикл ввода

Вопрос в том, где именно считываются вводимые данные? Хороший вопрос. Он позволяет выяснить, что на самом деле вводимые данные можно поместить в разные места: в начало цикла, в середину и в конец; однако большинство специалистов, занимающихся программированием игр, предпочитают размещать их непосредственно перед разделом *перемещения*. Таким образом, последний ввод игрока воздействует на текущий кадр.

На рис. 9.2 игровой цикл представлен более подробно: входные и прочие разделы показаны с большим количеством деталей. Не забывайте, что вы используете игровую консоль, в которой все ваши действия по выводу любого кадра должны выполняться внутри функции `Game_Main()`. По сути, весь ваш мир сосредоточен внутри этой единственной функции.

Теперь, когда я восстановил в вашей памяти информацию о том, где должны считываться вводимые данные, давайте посмотрим, как можно сделать это при помощи DirectInput!

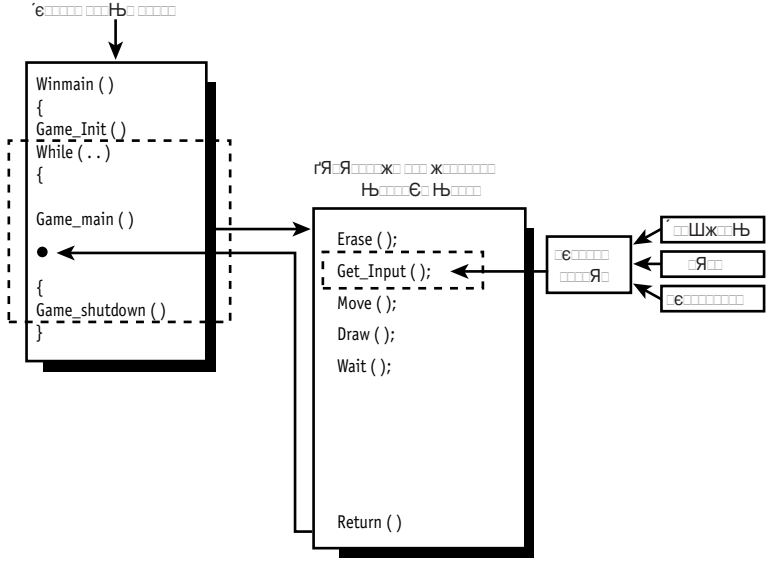


Рис. 9.2. Подробный цикл ввода

DirectInput: вступительная часть

По своей сути подсистема DirectInput — это столь же удивительная вещь, как и подсистема DirectDraw. Если бы не DirectInput, вы были бы вынуждены проводить дни у телефона, умоляя производителей устройств ввода, разбросанных по всему миру, предоставить вам необходимые драйверы (причем один для DOS, другой для Win16, третий для Win32 и т.д.). DirectInput устраняет все эти проблемы. Конечно, поскольку DirectInput была спроектирована компанией Microsoft, она создает ряд других проблем, но они, по крайней мере, локализованы в рамках одной компании!

DirectInput — это то же самое, что и DirectDraw, т.е. аппаратно-независимая виртуальная система ввода, позволяющая производителям аппаратных средств создавать стандартные и нестандартные устройства ввода, которые действуют как интерфейсы, причем одинаковым образом. Это очень хорошо, так как вам не нужен отдельный драйвер для каждого устройства ввода, которое может иметь ваш пользователь. Вы общаетесь с DirectInput стандартным способом, а драйвер этой подсистемы преобразует ваши обращения в аппаратно-зависимые вызовы так же, как это делает система DirectDraw.

На рис. 9.3 показана схема системы DirectInput и ее связь с драйверами аппаратного обеспечения и физическими устройствами ввода.

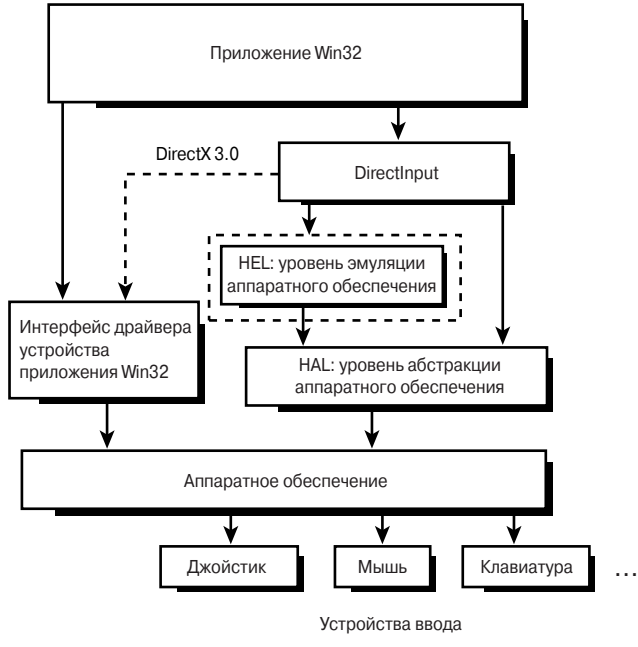


Рис. 9.3. Схема DirectInput на системном уровне

Как видно из рисунка, вы всегда изолированы от аппаратного обеспечения уровнем HAL (Hardware Abstraction Layer — уровень абстракции аппаратного обеспечения). В DirectInput, по сути, не требуется эмуляция, выполняемая HEL (Hardware Emulation Layer — уровень эмуляции аппаратного обеспечения), а потому этот уровень не играет здесь такой роли, как в DirectDraw. Давайте рассмотрим, что же поддерживает эта система.

Каждое отдельно существующее устройство ввода.

Это в значительной степени является правдой, если существует драйвер DirectInput для такого устройства. DirectInput может обращаться к нему, а следовательно, вы можете делать то же самое. Конечно, для этого производители аппаратных средств вынуждены писать драйвер, но это их проблемы. Учитывая все сказанное, вполне можно ожидать поддержки следующих устройств:

- клавиатура;
- мышь;
- джойстик;
- пульт ручного управления игрой*;
- игровая клавишная панель*;
- руль*;
- ручка управления полетами*;
- головной шлем виртуальной реальности*;
- объемный шар с шестью степенями свободы*;
- костюмы для киберсекса* (как только они попадут на рынок товаров массового производства в начале 2005 года :-))

С точки зрения DirectInput, все устройства, обозначенные звездочками, считаются джойстиками. Существует так много подклассов устройств ввода, похожих на джойстик, что DirectInput просто называет их одним словом — *устройства*. Каждое такое устройство может иметь один или несколько объектов ввода, которые могут быть перемещаемыми, вращательными, нажимными и т.д. Понятно? Например, джойстик, у которого две оси (X и Y) и два переключателя представлены четырьмя объектами ввода.

DirectInput практически не интересуется тем, является ли данное устройство джойстиком, поскольку оно вполне может представлять руль. DirectInput уделяет немного внимания подклассам. Все, что не является мышью или клавиатурой, представляет собой устройство, похожее на джойстик, независимо от того, держите ли вы его, вдавливаете, поворачиваете или нажимаете.

Чтобы различать все эти устройства, DirectInput заставляет производителя (а следовательно, и драйвер) задавать для каждого устройства GUID, представляющий его. Таким образом, для каждого устройства, которое существует или будет существовать, есть свое имя, и DirectInput может запрашивать систему о любом устройстве с данным именем. Если определенное устройство найдено, то оно представляет собой совокупность объектов ввода. Я так подробно рассматриваю эту тему, потому что она, как оказалось, сбивает людей с толку. Впрочем, мое объяснение, наверное, тоже. Так что давайте перейдем к следующему вопросу.

Компоненты системы DirectInput

DirectInput версии 8.0 содержит несколько COM-интерфейсов (что характерно для всех подсистем DirectX). Рассмотрим рис. 9.4, на котором показаны два основных интерфейса — `IDirectInput8` и `IDirectInputDevice8`.

Рассмотрим эти интерфейсы подробнее.

- `IDirectInput8`. Это основной COM-объект, который должен создаваться вами для запуска DirectInput. К счастью, существует функция `IDirectInput8Create()`, которая выполняет всю черную работу по настройке COM. Создав интерфейс `IDirectInput8` один раз, вы сможете обращаться к нему для настройки DirectInput, а также для создания и получения любого устройства ввода, которое может понадобиться для работы.

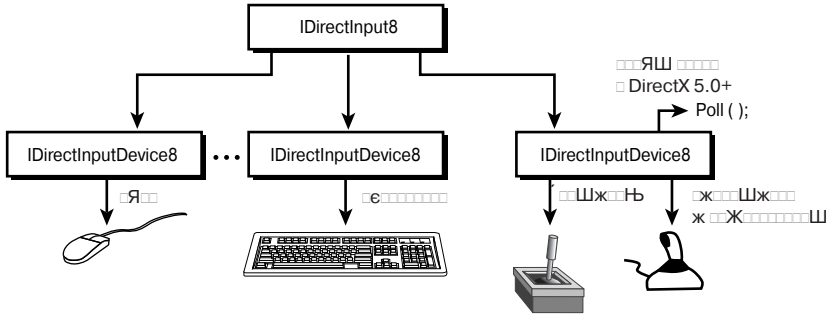


Рис. 9.4. Интерфейсы системы DirectInput

- IDirectInputDevice8. Этот интерфейс создается из основного интерфейса IDirectInput8 и представляет собой канал, используемый для связи с устройством (будь то мышь, клавиатура, джойстик или что-либо другое — все это устройства IDirectInput8). Он также поддерживает джойстики и устройства с виброотдачей, а кроме того, позволяет подключать опрашиваемые устройства (некоторые джойстики нуждаются в постоянном опросе).

Настройка DirectInput

Запуск и работа DirectInput, подключение одного или нескольких устройств и, наконец, получение данных от устройства (или устройств) требуют выполнения ряда действий. Вначале осуществляется настройка DirectInput, а затем каждого устройства ввода. Выполнение второй части практически одинаково для каждого устройства, поэтому ее можно обобщить. Итак, вот что вы должны сделать.

1. Создать основной интерфейс системы DirectInput — IDirectInput8 — посредством вызова функции IDirectInput8Create(), которая возвращает интерфейс IDirectInput8.
2. (Необязательно) Осуществить запрос глобально уникальных идентификаторов устройств. На этой стадии вы запрашиваете DirectInput на предмет получения устройств ввода, относящихся к следующим классам: клавиатура, мышь, джойстик или общее устройство (не указанное в списке). Эта операция выполняется при помощи функции обратного вызова для перечисления. Как правило, запрашивается перечисление всех устройств некоторого типа/подтипа. DirectInput фильтрует их посредством функции обратного вызова, и, таким образом, постепенно создается база данных глобально уникальных идентификаторов. К счастью, на практике эта проблема касается только устройств типа джойстиков, поскольку обычно считается, что в системе есть мышь и клавиатура, а для них существуют стандартные глобально уникальные идентификаторы. Немного позже вы узнаете, как эта операция работает применительно к джойстикам.
3. Каждое устройство, которое будет использоваться в вашем приложении, должно быть создано посредством обращения к функции CreateDevice() и передачи ей глобально уникального идентификатора. CreateDevice() представляет собой функцию интерфейса IDirectInput8, поэтому его необходимо получить до вызова функции. Кроме того, если вы не знаете GUID устройства, которое пытаетесь создать, то этому шагу должен предшествовать шаг 2. Имеются два встроенных глобально уникальных идентификатора: один — для клавиатуры, другой — для мыши.

   GUID_SysKeyboard. Определен глобально и всегда выполняет функцию GUID основной клавиатуры.

GUID_SysMouse. Определен глобальным образом и всегда выполняет функцию GUID основного устройства мыши.

СОВЕТ

> Недавно один хитроумный читатель прислал мне по электронной почте вопрос о том, как можно идентифицировать и использовать не одну, а несколько мышей. Вообще-то я не задумывался над этим, но если драйвер может поддерживать несколько мышей, то, работая под управлением DirectInput, вы должны иметь возможность его использовать. В этом случае для создания новой мыши необходимо запросить ее GUID.

4. Если вы создали устройство, следует задать его уровень взаимодействия, что выполняется посредством обращения к функции IDirectInputDevice8::SetCooperativeLevel(). Обратите внимание на синтаксис языка C++: это просто означает, что функция SetCooperativeLevel() представляет собой метод, или функцию интерфейса IDirectInputDevice8. Уровни взаимодействия очень похожи на соответствующие уровни DirectDraw, но их количество меньше. В примере, описывающем клавиатуру, они рассматриваются подробнее.
5. Задать для каждого устройства формат данных посредством вызова функции setDataFormat() интерфейса IDirectInputDevice8, что на практике немного сбивает с толку, но вполне логично с точки зрения идеи. Формат данных определяет, каким образом нужно формировать пакет данных для каждого события устройства. И это очень хорошо, так как такая характеристика просто придает дополнительную гибкость. Чтобы облегчить вашу участь, предлагаются некоторые предварительно заданные форматы данных, которые можно применять на практике; это выглядит вполне разумно, поскольку вы не обязаны самостоятельно задавать подобный формат.
6. Задать любые необходимые свойства устройства, используя функцию IDirectInputDevice8::setProperty(). Такая операция является контекстно-зависимой, т.е. одни устройства имеют определенные свойства, а другие нет. Таким образом, вы должны знать, что именно пытаетесь установить. Задавая только некоторые свойства из всего диапазона свойств устройства, учтите: все, что можно настроить, настраивается посредством функции SetProperty(). Как обычно, вызов выглядит ужасно, и я подробно расскажу о нем при рассмотрении примера, посвященного джойстику.
7. Получить каждое устройство посредством обращения к IDirectInputDevice8::Acquire(). При этом вызове устройство (или устройства) присоединяется или связывается с вашим приложением, а подсистеме DirectInput сообщается, что в будущем вы будете получать от него данные.
8. (Необязательно) Опросить устройство (или устройства) с помощью вызова IDirectInputDevice8::Poll(). Некоторые устройств ввода вместо генерации прерываний и хранения текущего состояния должны постоянно опрашиваться. В этот класс попадает ряд джойстиков, так что лучше опрашивать их постоянно, независимо от того, интересует ли его состояние нас или нет. Опрос не наносит никакого вреда и практически не привносит затрат (происходит обычный возврат из функции).
9. Прочитать данные каждого устройства вызовом функции IDirectInputDevice8::GetDeviceState(). Данные, возвращаемые в результате этого вызова, будут различными для каждого устройства, но сам вызов всегда абсолютно одинаков. В ходе выполнения этого вызова данные выбираются из устройства и помещаются в буфер, откуда их можно прочитать.

Вот и все. Выглядит весьма внушительно, но на самом деле это очень небольшая цена, которую вы платите за обладание доступом к любому устройству ввода, ничуть не беспокоясь о драйвере устройства для него.

Режимы получения данных

И наконец, хочу предупредить читателя о существовании *режима непосредственного получения данных* и *режима получения данных с буферизацией*. DirectInput может отправлять вам полученную информацию немедленно, а может буферизировать ее (с использованием временных меток в формате сообщений). Не могу похвастать длительным использованием буферизированных вводимых данных, так что для применения этого режима рекомендую заинтересованным читателям обратиться к документации DirectX SDK. В книге будет использоваться режим непосредственного получения данных, который задан по умолчанию.

Создание основного объекта DirectInput

Теперь рассмотрим, как создается основной COM-объект DirectInput — IDirectInput8, а затем перейдем к вопросам работы с клавиатурой, мышью и джойстиком.

Указатель интерфейса на основной объект DirectInput определяется в заголовочном файле DINPUT.H следующим образом:

```
LPDIRECTINPUT8 lpdi; // Основной интерфейс DirectInput
```

Для создания основного COM-интерфейса воспользуйтесь функцией DirectInput8Create():

```
HRESULT WINAPI DirectInput8Create(
  HINSTANCE hinst,           // Экземпляр приложения
  DWORD    dwVersion,        // Версия DirectInput
  REFIID   riidltf,          // идентификатор интерфейса
  LPVOID   *lplpDirectInput, // указатель на переменную
                             // для хранения указателя на интерфейс
  LPUNKNOWN punkOuter); // В книге всегда NULL
```

Вот описание используемых параметров.

hinst — дескриптор экземпляра вашего приложения. Это одна из немногих функций, которым требуется подобный дескриптор. Именно этот дескриптор передается функции WinMain() при запуске вашего приложения, так что сохраните его в глобальной переменной и используйте при необходимости.

dwVersion — константа, указывающая версию DirectInput, с которой должно быть совместимо ваше приложение. Если вы считаете, что игра должна работать на машинах с DirectX 3.0, — это ваше право, но мы будем использовать последнюю версию DirectInput, передавая в качестве данного параметра значение DIRECTINPUT_VERSION.

riidltf — константа, задающая версию создаваемого вами интерфейса. Обычно она равна IID_IDirectInput8.

lplpDirectInput — адрес переменной, в которой будет храниться полученный указатель на COM-интерфейс DirectInput.

И наконец, параметр punkOuter, как обычно, нами не рассматривается, здесь просто передается значение NULL.

DirectInput8Create() возвращает значение DI_OK в случае успешного завершения, а в случае неуспешного — некоторое иное значение. Как обычно, можно использовать макросы SUCCESS() и FAILURE() для проверки успешного завершения функции. Тем не менее проверка на равенство DI_OK вполне безопасна, и если вы пожелаете, то можете использовать именно ее.

Приведем пример создания основного объекта DirectInput.

```
#include "DINPUT.H"  // Требуется включение этого файла и
                     // и библиотек DINPUT.LIB и DINPUT8.LIB
```

```
// Остальные заголовочные файлы, определения и т.д.

// Глобальные переменные
LPDIRECTINPUT8  lpdi = NULL;  // используется для хранения
                              // указателя на COM-интерфейс

if (FAILED(DirectInput8Create(main_instance,
                  DIRECTINPUT_VERSION,
                  IID_ IDirectInput8,
                  (void **)&lpdi, NULL)))
  return(0);
```

НА ЗАМЕТКУ

Очень важно, чтобы вы включили в свое приложение заголовочный файл DINPUT.H, а также подключили библиотеку DINPUT.LIB; в противном случае компилятор и компоновщик не будут знать, что делать. Файл .LIB лучше подключить непосредственно в проект приложения. Задания пути в параметрах поиска библиотек обычно бывает недостаточно.

Если функция завершилась успешно, вы получаете указатель на основной объект DirectInput, который можно использовать для создания устройств.

Как всегда при работе с COM-объектами, по завершении работы приложения и после освобождения ресурсов вы должны освободить COM-объект вызовом функции Release().

```
// Завершение работы
lpdi->Release();
```

Если же вы хотите соблюсти все формальности, лучше записать так:

```
// Завершение работы
if (lpdi)
  lpdi->Release();
```

Конечно, это нужно делать только после освобождения созданных устройств. Необходимо помнить, что вызовы функции Release() должны осуществляться в порядке, обратном тому, в котором объекты создавались.

Клавиатура со 101 клавишей

Поскольку настройка одного устройства DirectInput похожа на настройку всех остальных устройств, сначала познакомимся со всеми деталями работы с клавиатурой, а затем перейдем к рассмотрению мыши и джойстика. Поэтому данный раздел следует прочитать с особым вниманием: содержащаяся в нем информация в равной степени применима ко всем остальным устройствам.

Создание устройства клавиатуры

При запуске и работе любого устройства первым шагом является создание этого устройства посредством вызова IDirectInput8::CreateDevice(). Необходимо помнить, что эта функция предоставляет интерфейс определенного устройства, которое вы запросили (в нашем случае — клавиатуру) и с которым можете затем работать. Давайте рассмотрим объявление функции:

```
HRESULT CreateDevice (
  REFGUID   rguid,          // GUID устройства
  LPDIRECTINPUTDEVICE8 *lplpDirectInputDevice,
                            // указатель на переменную для
                            // хранения указателя на интерфейс
  LPUNKNOWN pUnkOuter); // В книге всегда NULL
```

Достаточно просто, не правда ли? Первый параметр rguid представляет собой GUID создаваемого вами устройства. Вы можете либо запросить конкретный интересующий вас GUID, либо воспользоваться одним из идентификаторов по умолчанию для широко распространенных устройств:

GUID_SysKeyBoard — клавиатура;
GUID_SysMouse — мышь.

ВНИМАНИЕ

Не забывайте, что определения этих GUID находятся в заголовочном файле DINPUT.H, поэтому в код вашей программы следует обязательно включить этот файл. Кроме того, для доступа к определениям всех GUID необходимо использовать директиву препроцессора #define INITGUID в начале кода приложения, до директив #include. Следует также включить в программу заголовочный файл OBJBASE.H (это предпочтительнее, чем использовать библиотеку DXGUID.LIB). Если вы что-то забудете, то, чтобы освежить память, можете взглянуть на демонстрационные программы на прилагаемом компакт-диске.

Второй параметр указывает переменную-получатель нового интерфейса, а последний, как обычно, просто равен NULL. В случае успешного завершения функция возвращает DI_OK, и некоторое другое значение в случае неудачного завершения.

Теперь посмотрим, как нам создать устройство клавиатуры. Первое, что потребуется, — это переменная, которая будет содержать указатель на интерфейс, созданный в результате вызова. Все устройства имеют тип IDirectInputDevice8:

```
IDirectInputDevice8  lpdikey = NULL;
  // указатель на клавиатуру
```

Мы создадим устройство вызовом функции CreateDevice() из основного COM-объекта. Вот как выглядит соответствующий код, включающий создание основного COM-объекта и все необходимые директивы:

```
// Эта директива должна быть указана первой
#define INITGUID
// Включаемые заголовки
#include <OBJBASE.H>
#include "DINPUT.H"

// Глобальные переменные
LPDIRECTINPUT8 lpdi = NULL; // Указатель на COM-интерфейс
IDirectInputDevice8 lpdikey = NULL;
                              // Указатель на клавиатуру
if(FAILED(DirectInput8Create(main_instance,
                  DIRECTINPUT_VERSION,
                  IID_IDirectInput8,
                  (void**)&lpdi, NULL)))
  return(0)

// Создаем устройство клавиатуры
if (FAILED(lpdi->CreateDevice(GUID_SysKeyboard,
                   &lpdikey,NULL)))
  { /* ошибка */}

// Выполняем прочие действия
```

Теперь lpdikey указывает на устройство клавиатуры, и вы можете вызывать методы интерфейса для задания уровня взаимодействия, формата данных и т.п. Разумеется, когда

вы закончите всю работу с этим устройством, вы должны будете освободить его посредством вызова функции Release(). Однако этот вызов должен быть выполнен до того, как вы освободите основной объект DirectInput — lpdi, поэтому код, выполняемый при закрытии, должен выглядеть примерно так:

```
// Освободить все устройства
if (lpdikey)
  lpdikey->Release();

// Освобождение остальных устройств, джойстика, мыши и др.

// Освобождение основного COM-объекта
if (lpdi)
  lpdi->Release();
```

Определение уровня взаимодействия клавиатуры

Если устройство уже создано (в данном случае клавиатура), вы должны задать для него уровень взаимодействия; это делается точно так же, как и при работе с основным объектом системы DirectDraw. Однако в случае использования системы DirectInput выбор не так велик. В табл. 9.1 представлены возможные варианты уровня взаимодействия.

Таблица 9.1. Флаги взаимодействия для SetCooperativeLevel() подсистемы DirectInput

Значение	*Описание*
DISCL_BACKGROUND	Приложение может использовать устройство DirectInput, и когда оно находится в фоновом режиме, и когда оно активно и находится на переднем плане
DISCL_FOREGROUND	Приложение требует доступ переднего плана. Устройство автоматически отсоединяется при переходе приложения в фоновый режим
DISCL_EXCLUSIVE	После получения устройства никакое другое приложение не может запрашивать исключительный доступ к нему (однако в состоянии запросить неисключительный доступ)
DICSL_NONEXCLUSIVE	Приложение требует неисключительный доступ. Такой доступ к устройству не будет мешать другим приложениям, которые осуществляют доступ к тому же устройству

Ну как? Голова еще не болит? Фоновый режим, передний план, исключительный доступ... Однако, если прочитать определения несколько раз, становится понятно, каким образом действуют различные флаги. В общем, если установить DISCL_BACKGROUND, приложение будет получать вводимые данные независимо от того, активно оно или нет. При установке флага DISCL_FOREGROUND вводимые данные будут отправляться приложению только в том случае, когда оно работает на переднем плане.

Указание исключительного/неисключительного доступа управляет тем, будет ли ваше приложение иметь полный контроль над устройством (т.е. такой, при котором никакое другое приложение не сможет получить исключительного доступа). Например, мышь и клавиатура неявно являются исключительными устройствами: если ваше приложение получило их, никакое другое приложение не сможет использовать их до тех пор, пока не окажется в активном состоянии. При этом возникают некоторые парадоксы.

Во-первых, клавиатуру вы можете получить только в режиме неисключительного доступа, поскольку сама по себе операционная система Windows всегда имеет возможность вос-

принимать комбинации клавиш, включающие клавишу <Alt>. Во-вторых, если вы захотите, то можете получить мышь в режиме эксклюзивного доступа, но при этом будут потеряны стандартные сообщения, которые мышь передает вашему приложению (впрочем, может быть, это входило в ваши намерения), и исчезнет стандартный курсор мыши. Стоит ли волноваться? Вы, скорее всего, самостоятельно восстановите его для себя. И наконец, почти все джойстики должны быть получены в режиме исключительного доступа.

Таким образом, лучше всего указывать флаги DISCL_BACKGROUND|DISCL_NONEXCLUSIVE. Реальная необходимость в получении исключительного доступа возникает только в случае использования устройств с виброотдачей. Конечно, при задании такого значения флагов существует вероятность потерять устройство для какого-то другого приложения, которое потребует эксклюзивного доступа, когда станет активным. В таком случае вам придется заново получить устройство (вскоре будет рассматриваться и этот вопрос).

А сейчас просто зададим уровень взаимодействия при помощи функции IDirectInputDevice8::SetCooperativeLevel(...):

```
HRESULT SetCooperativeLevel (
  HWND hwnd,      // Дескриптор окна
  DWORD dwFlags); // Флаги взаимодействия
```

А вот как выглядит вызов, предназначенный для задания уровня взаимодействия вашей клавиатуры (это делается одинаково для всех устройств):

```
if (FAILED(lpdikey->SetCooperativeLevel(main_window_handle,
  DISCL_BACKGROUND | DISCL_NONEXCLUSIVE)))
  { /* ошибка */  }
```

Эта конструкция не будет работать только в том случае, если существует другое приложение, которое имеет исключительный доступ и является текущим. В таком случае следует просто подождать или попросить пользователя закрыть приложение, которое "прибрало к рукам" устройство ввода.

Задание формата данных клавиатуры

Следующий шаг при подготовке клавиатуры для передачи вводимых данных — задание формата данных, что осуществляется посредством вызова функции IDirectInputDevice8::SetDataFormat().

```
HRESULT SetDataFormat(
  LPCDIDATAFORMAT lpdf); // указатель на структуру формата
```

Этот единственный параметр только выглядит просто. На самом деле вот что он собой представляет:

```
//  формат данных системы  DirectInput
typedef struct
{
  DWORD dwSize;      // Размер структуры в байтах
  DWORD dwObjSize;  // Размер DIOBJECTDATAFORMAT в байтах
  DWORD dwFlags;     // Флаги: DIDF_ABSAXIS либо DIDF_RELAXIS
                     // для абсолютного или относительного
                     // сообщения
  DWORD dwDataSize; // Размер пакетов данных
  DWORD dwNumObjs; // Число объектов, определенных в
                     // следующем массиве объектов
  LPDIOBJECTDATAFORMAT  rgodf;
                      // Указатель на массив объектов
}  DIDATAFORMAT,  *LPDIDATAFORMAT;
```

Это действительно сложная для настройки структура, которая применима для решения всех ваших задач. Она, по сути, позволяет задавать способ форматирования данных (на уровне объекта устройства), получаемых от устройства ввода. Однако, к счастью, с DirectInput поставляются некоторые предопределенные форматы данных, которые можно применять почти в любой ситуации, и в своей работе вы будете просто использовать один из них. Рассмотрим табл. 9.2, в которой представлены эти форматы.

Таблица 9.2. Обобщенные форматы данных DirectInput

Значение	*Описание*
c_dfDIKeyboard	Обобщенная клавиатура
c_dfDIMouse	Обобщенная мышь
c_dfDIJoystick	Обобщенный джойстик
c_dfDIJoystick2	Обобщенное устройство с виброотдачей

Если вы зададите один из этих типов в качестве формата данных, система DirectInput будет возвращать каждый пакет данных в определенном формате (DirectInput и здесь предусматривает использование ряда встроенных форматов, облегчающих вашу работу; эти форматы представлены в табл. 9.3).

Таблица 9.3. Структуры данных DirectInput для работы с обобщенными форматами данных

Название	*Описание*
DIMOUSESTATE	Эта структура данных хранит сообщения от мыши
DIJOYSTATE	Эта структура данных хранит сообщения от джойстиков
DIJOYSTATE2	Эта структура данных хранит сообщения от устройств с виброотдачей

Реальные структуры будут представлены ниже, при рассмотрении мыши и джойстика. Однако вы, вероятно, озабочены тем, где же находится эта проклятая структура клавиатуры. Дело в том, что она настолько проста, что для нее не существует типа. Она задается простым 256-байтовым массивом, где каждый байт представляет одну клавишу, и в результате клавиатура выглядит как набор из 101 быстродействующего переключателя.

Таким образом, использование формата данных DirectInput по умолчанию, а также типы данных очень напоминают использование функции GetAsyncKeyState() Win32. Поэтому все, что нам нужно, — некоторый тип наподобие

```
typedef UCHAR _DIKEYSTATE[256];
```

СЕКРЕТ

Если в DirectX что-то отсутствует и я хочу создать "DirectX"-версию этого упущения, то, как правило, я должен создать отсутствующую структуру данных или функцию; но при этом вначале должен поставить знак подчеркивания, который и через полгода напомнит мне, что я сам ее создал[1].

Учитывая все сказанное, установим формат данных для нашей бедной маленькой клавиатуры:

[1] Профессионалы в программировании на языке C/C++ не рекомендуют этот метод; предпочтительнее использовать символы подчеркивания в конце идентификаторов. — *Прим. ред.*

```
// Определение формата данных
if (FAILED(lpdikey->SetDataFormat(&c_dfDIKeyboard)))
  { /* ошибка */ }
```

Заметьте, что я использовал оператор & для получения адреса глобальной переменной c_df_DIKeyboard, поскольку функция требует использования указателя на нее.

Захват клавиатуры

Вот мы и добрались до самого интересного. Уже почти все сделано. Создан основной COM-объект, создано устройство, установлен уровень взаимодействия и задан формат данных. Следующий шаг — это захват устройства у DirectInput. Для выполнения этой операции нужно воспользоваться методом IDirectInputDevice8::Acquire(), не имеющим параметров. Приведем пример:

```
// Получить клавиатуру
if (FAILED(lpdikey->Acquire()))
  { /* ошибка */ }
```

И это все, что нужно для подобной операции. Теперь вы готовы получать от устройства вводимые данные. Это нужно отпраздновать!

Получение данных от клавиатуры

Получение данных от любых устройств почти всегда выполняется одинаково, плюс-минус пара деталей, которые могут зависеть от конкретного устройства. В целом нужно сделать следующее.

1. (Необязательно) Опросить устройство типа джойстика.
2. Прочитать данные, получаемые из устройства, с помощью вызова функции IDirectInputDevice8::GetDeviceState().

СОВЕТ Помните, что любой метод, который можно вызвать с помощью интерфейса IDirectInputDevice, можно вызвать и посредством интерфейса IDirectInputDevice8.

Вот как выглядит функция GetDeviceState():

```
HRESULT GetDeviceState (
  DWORD cbData,     // Размеры структуры состояния
  LPVOID lpvData ); // Указатель на переменную для
                    // хранения полученных данных
```

Первый параметр представляет собой размер структуры получаемых данных, в которую будут помещены данные: 256 для данных клавиатуры, sizeof(DIMOUSESTATE) для мыши, sizeof(DIJOYSTATE) для простого джойстика и т.д. Второй параметр — это просто указатель, обозначающий то место, где должны быть сохранены данные. Теперь напишем код для чтения данных клавиатуры:

```
typedef UCHAR _DIKEYSTATE[256];
_DIKEYSTATE keystate; // здесь будут храниться данные
                      // клавиатуры
// Читаем данные клавиатуры
if (FAILED(lpdikey->GetDeviceState(sizeof(_DIKEYSTATE),
      (LPVOID)keystate)))
  { /* ошибка */ }
```

Разумеется, эту операцию можно выполнять один раз в каждом цикле игры — в самом его начале, до выполнения какой бы то ни было обработки.

Как только у вас будут данные, вы наверняка захотите выяснить, были ли нажаты какие-то клавиши и какие именно. Точно так же, как существуют константы для функции GetAsyncKeyState(), имеются соответствующие константы и для данных клавиатуры, которые определяют позиции внутри массива. Все они начинаются с символов DIK_ (я думаю, они обозначают словосочетание "DirectInput key") и определены в DINPUT.H. Табл. 9.4 содержит фрагмент списка этих констант (полный список можно найти в DirectX SDK).

Таблица 9.4. Константы состояния клавиатуры DirectInput

Символ	*Описание*
DIK_ESCAPE	<Esc>
DIK_0-9	Основная клавиатура: от 0 до 9
DIK_MINUS	<–>
DIK_EQUALS	<=>
DIK_BACK	<Backspace>
DIK_TAB	<Tab>
DIK_A-Z	Клавиши от A до Z
DIK_LBRACKET	<{>
DIK_RBRACKET	<}>
DIK_RETURN	<Enter> на основной клавиатуре
DIK_LCONTROL	Левая клавиша <Ctrl>
DIK_LSHIFT	Левая клавиша <Shift>
DIK_RSHIFT	Правая клавиша <Shift>
DIK_LMENU	Левая клавиша <Alt>
DIK_SPACE	Клавиша пробела
DIK_F1-15	Функциональные клавиши от 1 до 15
DIK_NUMPAD0-9	Клавиши цифровой клавиатуры
DIK_ADD	Знак + на цифровой клавиатуре
DIK_NUMPADENTER	<Enter> на цифровой клавиатуре
DIK_RCONTROL	Правая клавиша <Ctrl>
DIK_RMENU	Правая клавиша <Alt>
DIK_HOME	<Home>
DIK_UP	<↑>
DIK_PRIOR	<PgUp>
DIK_LEFT	<←>
DIK_RIGHT	<→>
DIK_END	<End>
DIK_DOWN	<↓>
DIK_NEXT	<PgDown>
DIK_INSERT	<Insert>
DIK_DELETE	<Delete>

Примечание. Элементы, выделенные жирным шрифтом, означают следование по порядку; например, DIK_0-9 означает, что здесь представлены константы DIK_0, DIK_1, DIK_2 и т.д.

Чтобы проверить, нажата ли какая-либо клавиша, нужно проверить бит 0x80 в 8-битовом байте, соответствующем данной клавише (иными словами — старший бит). Например, чтобы проверить, нажата ли клавиша <Esc>, нужно сделать следующее:

```
if (keystate[DIK_ESCAPE]&0x80)
  { // Клавиша нажата... */ }
else
  { /* ...или не нажата  */ }
```

СОВЕТ

Возможно, вы могли бы избежать использования оператора & и проверки бита, но Microsoft не гарантирует, что другие биты не будут установлены даже в том случае, когда клавиша не нажата. Проверка конкретного бита проводится в целях безопасности.

Я не люблю оператор & и всегда стараюсь упрятать его подальше, в макрос наподобие такого:

```
#define DIKEYDOWN(data,n)   (data[(n)] & 0x80 )
```

Тогда можно написать следующее:

```
if (DIKEYDOWN(keystate,DIK_ESCAPE))
  { /* Клавиша нажата */ }
```

И, конечно же, по окончании работы с клавиатурой необходимо вернуть ее при помощи вызова Unacquire() и освободить все захваченные ресурсы (включая основной COM-объект DirectInput) примерно так:

```
// Возврат клавиатуры
if (lpdikey)
  lpdikey->Unacquire();

//  Освобождение всех устройств
if (lpdikey)
  lpdikey->Release();

// Освобождение других устройств
...

// Освобождение основного COM-объекта
if (lpdi)
  lpdi->Release();
```

Функцию Unacquire() в первый раз я упоминаю только здесь, при освобождении объектов. Я посчитал уместным упомянуть ее именно здесь, но, конечно, если вы просто хотите отменить захват устройства, не освобождая его, то можете вызвать Unacquire() для этого устройства, а позже захватить его снова. Это может понадобиться, например, в том случае, когда вы хотите, чтобы другое приложение могло получить доступ к этому устройству, пока вы его не используете.

ВНИМАНИЕ

Разумеется, если вам захочется освободить клавиатуру, но оставить джойстик (или некоторую другую комбинацию устройств), не освобождайте основной COM-объект до тех пор, пока не будете готовы полностью отказаться от услуг DirectInput.

В качестве примера использования клавиатуры рассмотрим демонстрационную программу DEMO9_1.CPP на прилагаемом компакт-диске. На рис. 9.5 представлена копия экрана действующей программы. Эта программа применяет все описанные способы работы

с клавиатурой, что позволяет перемещать персонаж игры. При компиляции программы не забывайте включить в проект библиотеки DDRAW.LIB, DINPUT.LIB и WINMM.LIB. Кроме того, если взглянуть на раздел заголовочных файлов программы, можно заметить, что в нее включен файл T3DLIB1.H, так что в проект нужно включить еще и файл T3DLIB1.CPP. В конце главы описывается разработка библиотеки программ ввода (которую я уже создал и назвал T3DLIB2, но об этом позже).

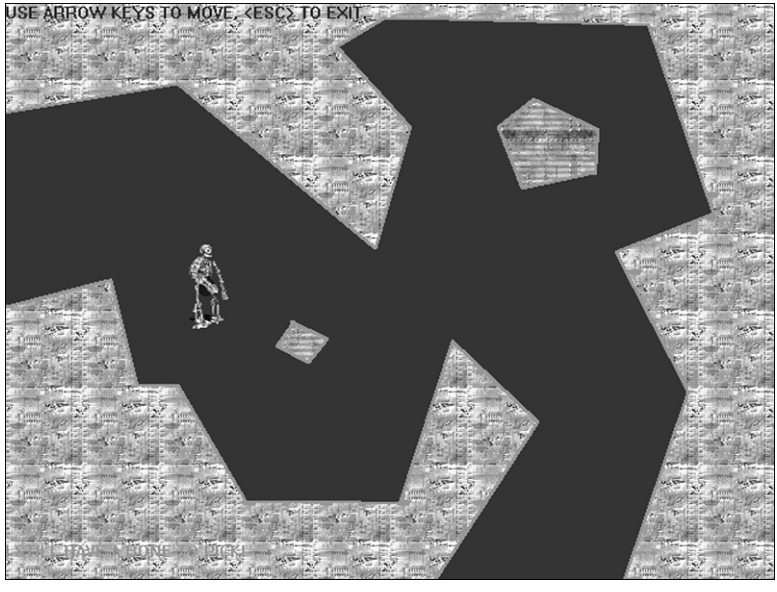


Рис. 9.5. Демонстрационная программа DEMO9_1.EXE

Повторный захват устройства

Я ненавижу разговоры о проблемах, которые могут появляться при работе с DirectX, потому что их слишком много. Эти проблемы нельзя назвать ошибками, это просто рабочие проявления в многозадачной операционной системе типа Windows. Одна такая проблема может возникнуть при работе с DirectInput в силу того, что устройство отбирается у вас или на него поступил запрос от другого приложения.

Вероятно, вы обладали устройством во время построения последнего кадра анимации, а теперь лишились его. Увы! Вы должны сами отслеживать такую ситуацию и запрашивать устройство заново. К счастью, существует способ выполнения подобной проверки, и он достаточно прост. Когда вы читаете данные из устройства, то проводите проверку, не поступил ли запрос на это устройство со стороны другого приложения. Если это действительно так, вы просто заново запрашиваете его и еще раз пытаетесь прочитать данные. Это можно сделать благодаря возвращаемому функцией GetDeviceState() коду ошибки, возможные значения которого представлены в табл. 9.5.

Таблица 9.5. Коды ошибок функции GetDeviceState()

Значение	*Описание*
DIERR_INPUTLOST	Устройство потеряло ввод и будет утрачено при следующем вызове
DIERR_INVALIDPARAM	Неверен один из параметров функции

Значение	*Описание*
DIERR_NOTACQUIRED	Вы полностью потеряли устройство
DIERR_NOTINITIALIZED	Устройство не готово
E_PENDING	Данные еще не доступны

Итак, единственное, что вам нужно сделать, — проверить возврат ошибки DIERR_INPUTLOST во время чтения, а затем попытаться заново захватить устройство, если произошла данная ошибка. Приведем пример:

```
HRESULT  result ; // Общий результат
while((result = lpdikey->GetDeviceState(
    sizeof(_DIKEYSTATE),
    (LPVOID)keystate)) == DIERR_INPUTLOST)
  {
    // Пытаемся захватить устройство еще раз
    if (FAILED(result=lpdikey->Acquire()))
    {
      break; // Серьезная ошибка
    } // if
  } // while

// Либо данные получены успешно, либо произошла ошибка.
// Проверяем, что же именно произошло.
if (FAILED(result))
  { /* ошибка */ }
```

СОВЕТ

Я показал вам пример повторного захвата клавиатуры, но вероятность такой ситуации очень мала. Чаще всего теряются устройства типа джойстика.

Мышеловка

Мышь представляет собой одно из самых удивительных устройств ввода, которые когда-либо были созданы. Можете ли вы себе представить, сколько людей смеялись над этим нелепым устройством? Думаю, теперь смеется изобретатель... Дело в том, что иногда отлично работают самые необычные вещи, и мышь является ярким примером тому. А теперь давайте займемся серьезной работой с мышью.

Стандартная мышь компьютера имеет две или три кнопки и две оси движения: X и Y. По мере перемещения мышь создает пакеты информации, описывающие изменения ее состояния, и отправляет их в компьютер через последовательный интерфейс. Затем эти данные обрабатываются драйвером и отправляются операционной системе Windows или подсистеме DirectX. Мышь для нас — черный ящик, так что давайте не вдаваться в детали ее устройства. Все, что нужно знать, — каким образом определить, когда она перемещается и когда нажата кнопка. DirectInput позволяет делать все это и даже больше.

Существует два способа связи с мышью: *абсолютный режим* и *относительный режим*. В абсолютном режиме мышь возвращает координаты своей позиции на экране, основываясь на положении курсора мыши. Таким образом, при разрешении экрана 640×480 можно ожидать, что позиция мыши должна изменяться в пределах (0–639, 0–479) (рис. 9.6).

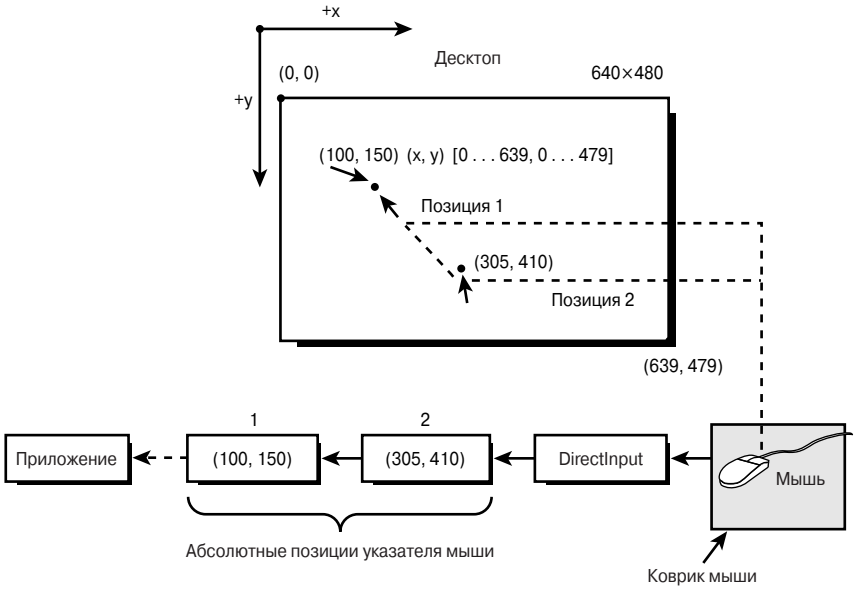


Рис. 9.6. Работа мыши в абсолютном режиме

В относительном режиме драйвер мыши отправляет не абсолютную позицию мыши, а относительные изменения ее координат за время такта системных часов, как показано на рис. 9.7. В действительности все мыши являются относительными: абсолютная позиция мыши отслеживается драйвером. Именно поэтому я намереваюсь изначально работать с мышью в относительном режиме, так как он является более гибким.

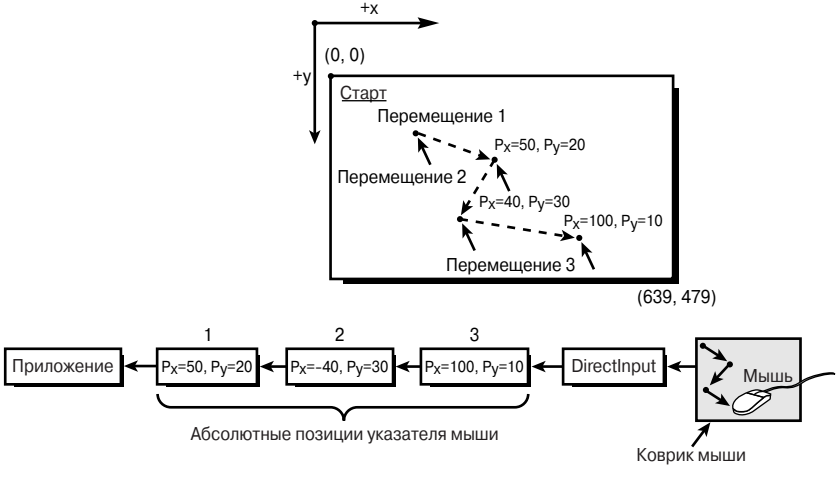


Рис. 9.7. Работа мыши в относительном режиме

Чтобы заставить мышь работать под управлением DirectInput, выполните следующее.

1. Создайте устройство мыши с помощью функции `CreateDevice()`.
2. Установите уровень взаимодействия с помощью функции `SetCooperativeLevel()`.

3. Задайте формат данных с помощью функции SetDataFormat().
4. Захватите мышь с помощью функции Acquire().
5. Прочитайте состояние мыши с помощью функции GetDeviceState().
6. Повторяйте шаг 5 до тех пор, пока не будет завершена работа с мышью.

НА ЗАМЕТКУ Если вам не знакомы эти операции, прочитайте, пожалуйста, предыдущий раздел, посвященный клавиатуре.

Создание устройства мыши

Сначала потребуется указатель интерфейса для хранения созданного устройства. Для этого используется указатель на интерфейс IDirectInputDevice8.

```
LPDIRECTINPUTDEVICE8 lpdimouse = NULL; // устройство мыши

// В предположении корректности lpdi

// Создаем устройство мыши
if (FAILED(lpdi->CreateDevice(GUID_SysMouse,
      &lpdimouse,NULL)))т
  { /* Ошибка */ }
```

Шаг 1 выполнен. Обратите внимание, что здесь использована константа GUID_SysMouse, что позволяет работать с устройством мыши, заданным по умолчанию.

Задание уровня взаимодействия для мыши

Теперь установим уровень взаимодействия.

```
if (FAILED(lpdimouse->SetCooperativeLevel(
      main_window_handle,
      DISCL_BACKGROUND|DISCL_NONEXCLUSIVE)))
  { /* ошибка */ }
```

Установка формата данных мыши

Установим формат данных мыши. Вспомните, что в DirectInput заранее предопределены стандартные форматы данных (представленные в табл. 9.2); один из них тот, который нужен нам, — c_dfDIMouse. Передавая его функции в качестве параметра, вы указываете формат данных, который будет использоваться при работе с мышью.

```
// Задание формата данных
if(FAILED(lpdimouse->SetDataFormat(&c_dfDIMouse)))
  { /* ошибка */ }
```

Сделаем небольшую остановку. При использовании формата данных клавиатуры — c_dfDIKeyboard — возвращаемая структура данных представляла собой массив, состоящий из 256 элементов типа UCHAR. Однако при работе с мышью формат данных должен быть более приспособлен для этого устройства. Обратимся к табл. 9.3. Структура данных для работы с мышью — DIMOUSESTATE — выглядит следующим образом:

```
// Структура данных для работы с мышью
typedef struct _DIMOUSESTATE
  {
    LONG lX;  // ось X
    LONG lY;  // ось Y
```

```
    LONG lX;  // ось Z (в большинстве случаев - колесо)
    BYTE rgbButtons [4]; // Кнопки мыши; старший бит
                  // указывает нажатое состояние
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

При обращении к функции GetDeviceState() для получения состояния устройства вы получаете эту структуру. Здесь нет ничего удивительного — все обстоит именно так, как и должно быть.

Захват мыши

Следующий шаг — захват мыши посредством вызова функции Acquire(). Это делается следующим образом:

```
// Захватить мышь
if(FAILED(lpdimouse->Acquire()))
  { /* ошибка */ }
```

Как видите, все предельно просто.

Чтение данных мыши

Итак, вы создали устройство мыши, установили уровень взаимодействия и формат данных и захватили устройство. Теперь можно воспользоваться добытыми трофеями. Для этого нужно получить данные от мыши при помощи вызова функции GetDeviceState(). При этом, разумеется, следует передать правильные параметры, соответствующие формату данных c_dfDIMouse и структуре данных, используемой для передачи информации — DIMOUSESTATE. Давайте теперь прочитаем данные мыши:

```
DIMOUSESTATE mousestate; // Для хранения данных мыши

// ... где-то в основном цикле ...

// Прочитать состояние мыши
if (FAILED(lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE),
      (LPVOID)mousestate)))
  { /* ошибка */ }
```

СЕКРЕТ

Обратите внимание на интеллектуальность функции GetDeviceState(). Вместо множества функций используется одна, параметрами которой являются размер и указатель, что позволяет работать с любыми форматами данных — как с теми, которые уже существуют в настоящий момент, так и с теми, которые могут понадобиться в будущем. Этот замечательный стиль программирования стоит того, чтобы его запомнить.

Теперь, когда у вас есть данные мыши, можно поработать с ней. Предположим, вы хотите переместить объект, основываясь на движении мыши. Если игрок двигает мышь влево, нужно, чтобы объект переместился влево на такое же расстояние. Кроме того, если пользователь нажимает левую кнопку мыши, должен запускаться реактивный снаряд, а правая кнопка должна осуществлять выход из программы. Напишем основной код.

```
// Безусловно, вы должны сделать все остальные шаги...

// Определения
# define MOUSE_LEFT_BUTTON   0
# define MOUSE_RIGHT _BUTTON  1
```

```
# define MOUSE_MIDDLE _BUTTON  2 // (Чаще всего)

// Глобальные переменные
DIMOUSESTATE mousestate;           // Данные мыши
int  object_x = SCREEN_CENTER_X,  // Изначально помещаем
    object_y = SCREEN_CENTER_y;    // объект в центр

// ...где-то внутри основного цикла ...

// Прочитать состояние мыши
if (FAILED(lpdimouse->GetDeviceState(sizeof (DIMOUSESTATE),
        (LPVOID)mousestate)))
   { /* ошибка  */ }

// Переместить объект
object_x += mousestate.lX;
object_y += mousestate.lY;

// Проверить кнопки
if (mousestate.rgbButtons[MOUSE_LEFT_BUTTON]&0x80)
   { /* Использовать оружие  */ }
else
if (mousestate.rgbButtons[MOUSE_RIGHT_BUTTON]&0x80)
   { /* Отправить сообщение о выходе из программы  */ }
```

Освобождение мыши

По окончании работы с мышью прежде всего следует отменить ее захват посредством вызова функции Unacquire(), а затем, как обычно, освободить устройство. Приведем пример соответствующего кода.

```
// Отменить захват мыши
if (lpdimouse)
   lpdimouse->Unacquire();

// Освободить мышь
if (lpdimouse)
   lpdimouse->Release();
```

Пример работы с мышью содержится в демонстрационной программе под названием DEMO9_2.CPP (16-битовая версия — в файле DEMO9_2_16B.CPP). Для компиляции этого приложения вы должны не забыть подключить к проекту библиотеки DDRAW.LIB, DINPUT.LIB и WINMM.LIB, а также библиотеку T3DLIB1.CPP. На рис. 9.8 показана копия экрана программы.

Работа с джойстиком

Пожалуй, джойстик — самое сложное из всех устройств, поддерживаемых DirectInput. Термин *джойстик* на самом деле охватывает всевозможные устройства, не относящиеся к клавиатуре или мыши. Однако, чтобы быть последовательным, я собираюсь вначале уделить внимание устройствам, которые выглядят как джойстик или пульт ручного управления электронной игрой, например Microsoft Sidewinder, Microsoft Gamepad, Gravis Flight Stick и т.п.

Рис. 9.8. DEMO9_2.EXE в действии

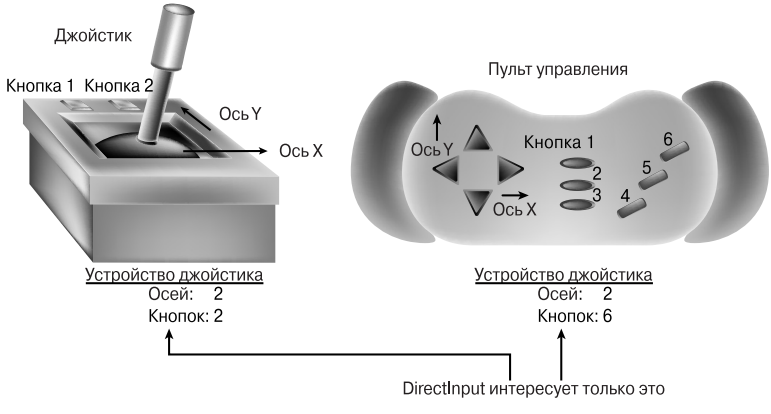


Рис. 9.9. Устройства DirectInput представляют собой коллекции объектов устройств

Прежде чем начать рассмотрение этого вопроса, взгляните на рис. 9.9. Здесь изображены джойстик и панель управления. Оба эти устройства при работе под управлением DirectInput считаются джойстиками. В отдельный класс выделены лишь *устройства с виброотдачей* (forced feedback), но мы не будем их рассматривать.

В любом случае мне хочется высказать одно замечание по поводу джойстика и игрового пульта: когда дело касается DirectInput, эти устройства считаются одинаковыми. Оба они представляют собой коллекцию осей, переключателей и ползунков. Просто оси джойстика имеют множество положений (непрерывных), а в игровом пульте предусмотрены фиксированные или крайние позиции. И главное в том, что каждое устройство представляет собой коллекцию *объектов устройств*, или *предметов устройств*, или *объектов ввода* (применение конкретного термина зависит от вас или от той книги, которую вы используете). Все они просто являются устройствами ввода, которые могут быть представлены одним и тем же физическим фрагментом аппаратного обеспечения. Понятно? Надеюсь, что да.

Шаги, выполняемые при настройке устройств типа джойстиков, такие же, как и при настройке клавиатуры и мыши, за исключением пары дополнительных шагов.

1. Создайте устройства с помощью функции CreateDevice().
2. Установите уровень взаимодействия с помощью функции SetCooperativeLevel().
3. Задайте формат данных с помощью функции SetDataFormat().
4. Задайте диапазон джойстика, мертвую зону и другие свойства с помощью функции SetProperties() (это новый шаг по сравнению с предыдущими устройствами).
5. Захватите джойстик с помощью функции Acquire().
6. Опросите джойстик с помощью функции Poll(). Этот шаг гарантирует, что джойстики, драйверы которых работают без использования прерываний, вернут корректные данные при вызове функции GetDeviceState() (это тоже новый шаг по сравнению с предыдущими устройствами).
7. Прочитайте состояние джойстика с помощью функции GetDeviceState().
8. Повторяйте шаг 7 до тех пор, пока не будет завершена работа с джойстиком.

Перечисление джойстиков

Мне никогда не нравилось объяснять концепции функций обратного вызова и функций перечисления, так как они кажутся мне весьма сложными. Но я надеюсь, что вы сами знакомы с функциями этого типа, поскольку программирование в DOS также использовало их. Если вы только начинаете изучение программирования в операционной системе Windows, эта тема может показаться вам слишком сложной; но однажды пройдя через нее, вам уже никогда не придется беспокоиться по этому поводу.

По своей сути функция обратного вызова чем-то похожа на WinProc() в ваших Windows-программах. Она представляет собой функцию, которую в некоторых ситуациях Windows вызывает, выполняя ваш запрос. На рис. 9.10 показана схема концепции обратного вызова.

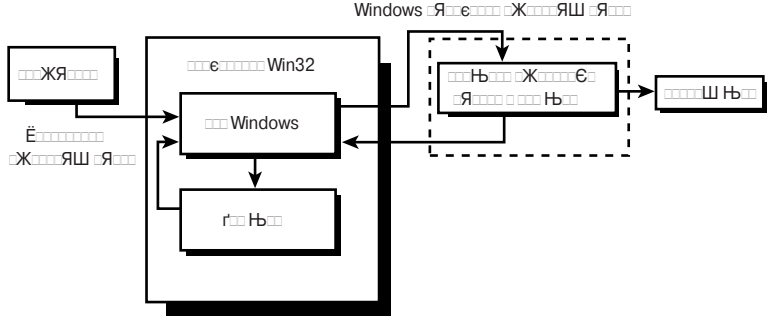


Рис. 9.10. Функция обратного вызова

Win32/DirectX использует функции обратного вызова и при перечислении. *Перечисление* означает, что Windows (или DirectInput в нашем случае) требуется просканировать системный реестр или какую-то базу данных в поиске, например, подключенных и доступных в системе джойстиков определенного вида.

Решить эту задачу можно двумя способами.

- Вызвать функцию системы DirectInput, которая создаст для вас список и сохранит его в некоторой структуре данных, а позже вы разберете и проанализируете полученную информацию.

НА ЗАМЕТКУ

Существует ряд других подтипов, которые не вошли в этот список. Главное в том, что поиск DirectInput может быть как общим, так и специализированным. В книге мы собираемся использовать в качестве значения dwDevType просто DIDEVTYPE_JOYSTICK, поскольку хотим работать с обычным джойстиком.

Следующий параметр в EnumDevices() — это указатель на функцию обратного вызова, которую DirectInput будет вызывать для каждого найденного ею устройства. Немного позже я покажу вид этой функции. Следующий параметр — pvRef — представляет собой 32-битовый указатель, который будет передан функции обратного вызова. Таким образом, при обратном вызове можно изменять значение указываемой величины или использовать ее для возврата данных, которые при этом не придется возвращать через глобальную переменную.

И наконец, dwFlags указывает, каким образом функция перечисления должна осуществлять сканирование. Должна ли она выполнять сканирование для поиска всех устройств, или только тех, которые подключены, или, например, только устройств с виброотдачей? В табл. 9.8 представлены коды сканирования, позволяющие управлять перечислением.

Таблица 9.8. Коды управления сканированием при перечислении

Значение	*Описание*
DIEDFLL_ALLDEVICES	Сканирует все установленные устройства, даже если они в данный момент не подсоединены
DIEDFLL_ATTACHEDONLY	Сканирует устройства, которые установлены и подсоединены
DIEDFLL_FORCEFEEDBACK	Сканирует устройства с виброотдачей

ВНИМАНИЕ

Следует использовать значение DIEDFL_ATTACHEDONLY, поскольку нет никакого смысла в подключении к устройству, которое не подсоединено к компьютеру.

Теперь рассмотрим более подробно функцию обратного вызова. Работа EnumDevices() заключается в том, что она в цикле осуществляет обратный вызов для каждого найденного ею устройства, как показано на рис. 9.11. Это означает, что если на компьютере установлен ряд устройств, то функция обратного вызова будет выполняться много раз.

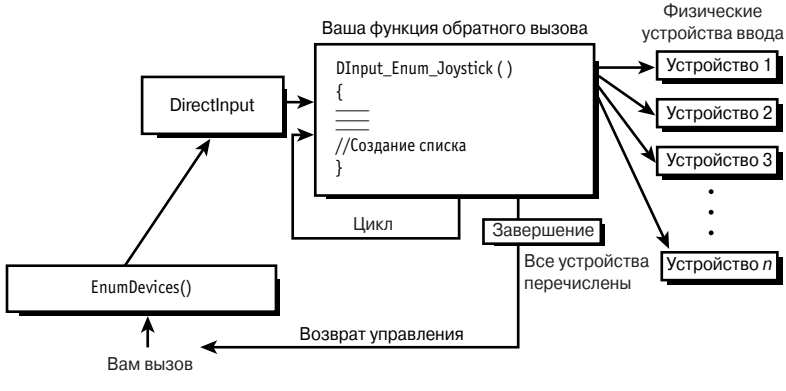


Рис. 9.11. Схема выполнения перечисления устройств

Таким образом, ваша функция обратного вызова может заносить все эти устройства в таблицу или куда-нибудь еще, где их можно будет просмотреть после завершения работы

НА ЗАМЕТКУ

Существует ряд других подтипов, которые не вошли в этот список. Главное в том, что поиск DirectInput может быть как общим, так и специализированным. В книге мы собираемся использовать в качестве значения dwDevType просто DIDEVTYPE_JOYSTICK, поскольку хотим работать с обычным джойстиком.

Следующий параметр в EnumDevices() — это указатель на функцию обратного вызова, которую DirectInput будет вызывать для каждого найденного ею устройства. Немного позже я покажу вид этой функции. Следующий параметр — pvRef — представляет собой 32-битовый указатель, который будет передан функции обратного вызова. Таким образом, при обратном вызове можно изменять значение указываемой величины или использовать ее для возврата данных, которые при этом не придется возвращать через глобальную переменную.

И наконец, dwFlags указывает, каким образом функция перечисления должна осуществлять сканирование. Должна ли она выполнять сканирование для поиска всех устройств, или только тех, которые подключены, или, например, только устройств с виброотдачей? В табл. 9.8 представлены коды сканирования, позволяющие управлять перечислением.

Таблица 9.8. Коды управления сканированием при перечислении

Значение	*Описание*
DIEDFLL_ALLDEVICES	Сканирует все установленные устройства, даже если они в данный момент не подсоединены
DIEDFLL_ATTACHEDONLY	Сканирует устройства, которые установлены и подсоединены
DIEDFLL_FORCEFEEDBACK	Сканирует устройства с виброотдачей

ВНИМАНИЕ

Следует использовать значение DIEDFL_ATTACHEDONLY, поскольку нет никакого смысла в подключении к устройству, которое не подсоединено к компьютеру.

Теперь рассмотрим более подробно функцию обратного вызова. Работа EnumDevices() заключается в том, что она в цикле осуществляет обратный вызов для каждого найденного ею устройства, как показано на рис. 9.11. Это означает, что если на компьютере установлен ряд устройств, то функция обратного вызова будет выполняться много раз.

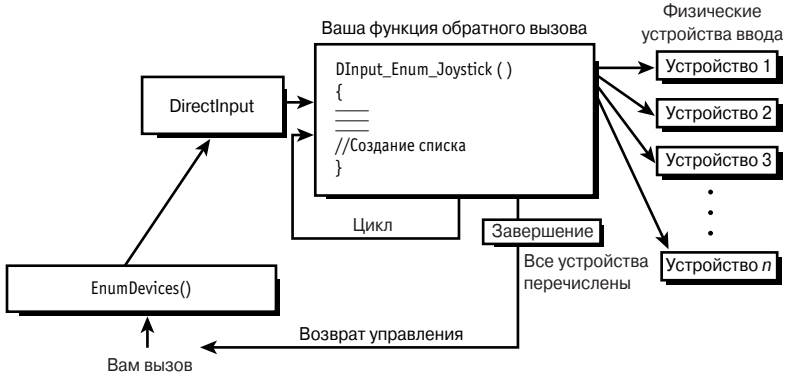


Рис. 9.11. Схема выполнения перечисления устройств

Таким образом, ваша функция обратного вызова может заносить все эти устройства в таблицу или куда-нибудь еще, где их можно будет просмотреть после завершения работы

функции EnumDevices(). С учетом сказанного рассмотрим обобщенный прототип функции обратного вызова, совместимой с DirectInput.

```
BOOL CALLBACK EnumDevsCallback(
  LPDIDEVICEINSTANCE lpddi,
                     // Указатель DirectInput, содержащий
                     // информацию о найденном устройстве
  LPVOID data );  // Указатель, переданный в параметре
                     // pvRef функции EnumDevices()
```

Все, что вам нужно сделать, — написать собственную функцию с данным прототипом и передать ее в качестве параметра lpCallback функции EnumDevices(). Имя этой функции можно выбирать по своему усмотрению.

Что именно вы поместите внутри этой функции, зависит, конечно, от вас. Вероятно, вы захотите записать или организовать в виде списка имена всех устройств и их GUID. Не забывайте, что ваша функция будет вызываться для каждого найденного устройства ровно один раз. Затем, имея на руках список, вы можете выбрать устройство самостоятельно или позволить сделать это пользователю, а затем использовать связанный с этим устройством GUID для создания устройства.

Кроме того, DirectInput позволяет продолжить перечисление или прекратить его в любое время; этим управляет значение, которое возвращает функция обратного вызова. Она может возвратить одну из следующих констант:

- DIENUM_CONTINUE — продолжить перечисление;
- DIENUM_STOP — остановить перечисление.

Итак, если вы зададите DIENUM_STOP в качестве возвращаемого значения функции обратного вызова, будет перечислено только одно устройство (даже если на самом деле их больше). Ограниченный объем книги не позволяет привести здесь функцию, которая помещает в список все GUID устройств, но я собираюсь продемонстрировать вам одну функцию, которая находит первое устройство и устанавливает его.

Однако сначала кратко рассмотрим структуру данных DIDEVICEINSTANCE, которая передается вашей функции обратного вызова для каждого элемента перечисления. Она содержит много интересной информации об устройстве.

```
typedef struct _DIDEVICEINSTANCE
{
  DWORD dwSize;       // Размер структуры
  GUID  guidInstance; // GUID устройства
  GUID  guidProduct;  // Общий GUID типа устройства
  DWORD dwDevType;    // Тип устройства (соответствует
                      // данным из табл. 9.6)
  CHAR  tszInstanceName[MAX_PATH];
                      // Обобщенное имя экземпляра
                      // устройства джойстика, например
                      // "joystick 1"
  CHAR  tszProductName[MAX_PATH];
                     // Имя изделия, например
                     // "Microsoft Sidewinder Pro"
  GUID  guidFFDriver; // GUID драйвера устройства
                      // с виброотдачей
  WORD  wUsagePage; // Расширение (не рассматривается)
  WORD  wUsage;     // Расширение (не рассматривается)
}  DIDEVICEINSTANCE, *LPDIDEVICEINSTANCE;
```

В большинстве случаев представляют интерес только поля tszProductName и guidInstance. Используя приведенное определение структуры DIDEVICEINSTANCE, вот как можно записать функцию обратного вызова для получения GUID первого же устройства джойстика.

```
BOOL CALLBACK DInput_Enum_Joysticks(
  LPCDIDEVICEINSTANCE lpddi,
  LPVOID          guid_ptr )
{
// Эта функция останавливает перечисление джойстиков
// после первого же найденного экземпляра и возвращает
// его GUID. Обратите внимание на использование приведения
// типа указателя
  *(GUID*)guid_ptr = lpddi->guidInstance;

  // Копирование имени продукта в глобальную переменную
  strcpy(joyname, (char*)lpddi->tszProductName);

  // Прекращение процесса перечисления
  return(DIENUM_STOP ) ;
} // DInput_Enum_Joysticks
```

Чтобы использовать эту функцию для поиска первого джойстика, можно воспользоваться кодом наподобие следующего:

```
char joyname[80];  // Область для имени джойстика
GUID joystickGUID;  // GUID джойстика

// Перечисление устройств джойстиков с использованием
// функции обратного вызова DInput_Enum_Joysticks
if (FAILED(lpdi->EnumDevices(
       DIDEVTYPE_JOYSTICK,     // Только джойстики
       DInput_Enum_Joysticks,  // Функция перечисления
       &joystickGUID,          // GUID
       DIEDFL _ATTACHEDONLY)))
  { /* ошибка */ }
// Обратите внимание на то, что выполнялось сканирование
// только присоединенных джойстиков
```

При использовании реального программного продукта у вас может возникнуть желание продолжить выполнение функции перечисления до тех пор, пока она не найдет все устройства, и только потом, на стадии настройки, позволить игроку выбрать устройство из списка. После этого связанный с выбранным устройством GUID используется для создания данного устройства, что и является следующим шагом в работе программы.

Создание джойстика

Имея "на руках" GUID создаваемого устройства, нужно, как и обычно, вызвать функцию CreateDevice(). Предполагая, что имело место обращение к функции EnumDevices() и GUID устройства был сохранен в переменной joystickGUID, приведем пример создания устройства джойстика.

```
LPDIRECTINPUTDEVICE8 lpdijoy; // Интерфейс джойстика
// Создание джойстика при помощи GUID
if (FAILED(lpdi->CreateDevice(joystickGUID,
                  &lpdijoy,NULL)))
  { /* ошибка */ }
```

НА ЗАМЕТКУ

В созданных демонстрационных программах я стремился использовать временные указатели на старые интерфейсы, чтобы получить доступ к самым последним интерфейсам. Таким образом, в демонстрационной программе я использую временный указатель, запрашиваю последний интерфейс и вызываю не lpdijoy2, а lpdijoy. Все интерфейсы, упоминаемые в этой книге, являются самыми последними версиями интерфейсов.

Задание уровня взаимодействия джойстика

Эта операция выполняется точно так же, как соответствующие операции для мыши и клавиатуры. Однако, если у вас есть устройство с виброотдачей, вам может потребоваться получить к нему исключительный доступ. Вот как это делается:

```
if (FAILED(lpdijoy->SetCooperativeLevel(
      main_window_handle,
      DISCL_BACKGROUND|DISCL_EXCLUSIVE)))
  { /* ошибка */ }
```

Задание формата данных

Теперь займемся форматом данных. Как и в случае с мышью и клавиатурой, воспользуемся стандартным форматом данных (из тех, что представлены в табл. 9.2). Один из форматов, которые могут вам понадобиться, — это c_dfDIJoystick (ci_dfDIJoystick2 предназначен для устройств с виброотдачей). Его и нужно передать функции в качестве параметра формата данных.

```
// Задать формат данных
if (FAILED(lpdijoy->SetDataFormat(&c_dfDIJoystick)))
  { /* ошибка */ }
```

Как и в случае с мышью, для работы вам понадобится специализированная структура данных, в которой будут храниться данные о состоянии устройства джойстика. Возвращаясь к табл. 9.3, отметим, что эта структура данных называется DIJOYSTATE (или DIJOYSTATE2 для устройств с виброотдачей) и имеет вид

```
// Обобщенная структура данных виртуального джойстика

typedef struct _DIJOYSTATE
{
  LONG lX;  // Ось X джойстика
  LONG lY;  // Ось Y  джойстика
  LONG lZ;  // Ось Z джойстика
  LONG lRx; // x-поворот джойстика (контекстно-зависимый)
  LONG lRy; // y-поворот джойстика (контекстно-зависимый)
  LONG lRz; // z-поворот джойстика (контекстно-зависимый)
  LONG  rglSlider[2];       // Устройства управления типа
                            // ползунков, педалей и т.п.
  DWORD rgdwROV[4];     // Элементы управления точкой
                            // обозрения, до 4 шт.
  BYTE  rgbButtons[32]; // 32 стандартные
                            // быстродействующие кнопки
} DIJOYSTATE, *LPDIJOYSTATE;
```

Как видите, у этой структуры много полей данных. Такой обобщенный формат данных весьма универсален. Я сомневаюсь, что вам понадобится когда-либо создавать собственный формат данных, поскольку мне попадалось немного джойстиков, которые об-

ладали бы бо́льшими возможностями, чем способен описать этот формат. Назначение полей пояснено комментариями. Для осей задается диапазон, а кнопки обычно представляют собой кнопки мгновенного действия; при этом установленный старший бит свидетельствует о том, что кнопка нажата.

Итак, именно эта структура возвращается при вызове функции GetDeviceState() для джойстика.

Сделано почти все. Есть только одна дополнительная деталь, которую нужно учесть: что именно означают те значения, которые возвращаются в этой структуре. Кнопка есть кнопка — она либо нажата, либо нет, но такие элементы, как, например, lX, lY и lZ, могут быть разными у различных производителей. Подсистема DirectInput позволяет масштабировать их, задавая определенный диапазон для любого случая, так что входная логика вашей игры всегда сможет работать с одними и теми же числами. Давайте посмотрим, каким образом задаются эти и другие свойства джойстика.

Задание входных свойств джойстика

Поскольку джойстик по своей сути является аналоговым устройством, движение ручки управления имеет ограниченный диапазон. Проблема состоит в том, что нужно задать определенные значения диапазона с тем, чтобы ваш код игры мог корректно интерпретировать движение ручки. Иными словами, когда вы запрашиваете позицию джойстика и он отвечает lX = 2000, lY = -3445, что это означает? Вы не можете интерпретировать данные, поскольку у вас нет системы отсчета, необходимой для полной ясности.

Как минимум, нужно установить диапазоны каждой аналоговой оси, которая будет считываться. Например, вы можете прийти к решению, что для осей X и Y надо установить диапазоны от -1000 до 1000 и от -2000 до 2000 соответственно, а может быть, задать для обеих диапазон от -128 до 128 с тем, чтобы можно было разместить значение в пределах одного байта. Что́ бы вы ни решили, сделать что-то все равно придется. В противном случае после прочтения данных у вас не будет никакого способа понять их смысл.

Задание любого свойства джойстика, включая диапазоны, выполняется при помощи функции SetProperty(). Ее прототип выглядит так:

```
HRESULT SetProperty(
  REFGUID rguidProp,          // GUID изменяемого свойства
  LPCDIPROPHEADER pdiph); // Указатель на структуру
                              // заголовка свойства, содержащую
                              // подробную информацию
```

Функция SetProperty() применяется для задания различных свойств, например относительного или абсолютного формата данных, диапазона каждой оси, мертвой зоны (или мертвой полосы, т.е. нейтральной области) и т.д. Использовать функцию SetProperty() крайне сложно из-за разнообразия природы всех констант и вложенных структур данных.

Достаточно сказать, что лучше не вызывать функцию SetProperty(), если в этом нет крайней необходимости. Подавляющее большинство значений по умолчанию будут прекрасно работать.

К счастью, чтобы заставить заработать джойстик, нужно задать только диапазоны по осям X и Y (и, возможно, мертвую зону) — и это все, что я собираюсь вам показать. Если вы заинтересованы в получении более подробной информации на эту тему, советую обратиться к DirectX SDK. Тем не менее показанный далее код поможет вам разобраться с тем, как в случае необходимости задаются и другие свойства. Структура, требующаяся для задания этих свойств, имеет вид

```
typedef struct _DIPROPRANGE
{
  DIPROPHEADER diph;
  LONG       lMin;
  LONG       lMax;
} DIPROPRANGE, *LPDIPROPRANGE;
```

Как видите, в эту структуру вложена другая структура — DIPROPHEADER:

```
typedef struct _DIPROPHEADER
{
  DWORD  dwSize;
  DWORD  dwHeaderSize;
  DWORD  dwObj;
  DWORD  dwHow;
} DIPROPHEADER, *LPDIPROPHEADER;
```

Обе структуры имеют миллиард способов установки, поэтому если данная тема вызывает у вас интерес, рекомендую обратиться к DirectX SDK. Для простого перечисления всех разнообразных флагов, которые вы можете передать, потребовалось бы более 10 страниц! Но это не помешает привести пример кода, предназначенного для задания диапазонов осей.

```
// Здесь хранятся данные для изменения свойств
DIPROPRANGE joy_axis_range;

// Задаем диапазон для оси x: от -1024 до 1024
joy_axis_range.lMin = -1024;
joy_axis_range.lMax =  1024;

joy_axis_range.diph.dwSize      = sizeof(DIPROPRANGE);
joy_axis_range.diph.dwHeaderSize = sizeof(DIPROPHEADER);

// Здесь хранится объект, подлежащий изменению
joy_axis_range.diph.dwObj       = DIJOFS_X;

// Этот объект может быть одним из:
// DIJOFS_BUTTON(n)  — для кнопок
// DIJOFS_POV(n)     — для индикаторов обзора
// DIJOFS_RX         — для поворота вокруг оси x
// DIJOFS_RY         — для поворота вокруг оси y
// DIJOFS_RZ         — для поворота вокруг оси z (руль)
// DIJOFS_X          — для оси x
// DIJOFS_Y          — для оси y
// DIJOFS_Z          — для оси z
// DIJOFS_SLIDER (n) — для любого ползунка
// Метод доступа к объекту всегда использует этот способ
joy_axis_range.diph.dwHow = DIPH_BYOFFSET;

// Задание свойства
lpdijoy->SetProperty(DIPROP_RANGE,&joy_axis_range.diph);

// Свойства для оси Y
joy_axis_range.lMin = -1024;
joy_axis_range.lMax =  1024;
joy_axis_range.diph.dwSize      = sizeof(DIPROPRANGE );
```

```
joy_axis_range.diph.dwHeaderSize = sizeof(DIPROPHEADER );
joy_axis_range.diph.dwObj      = DIJOFS_Y;
joy_axis_range.diph.dwHow      = DIPH_BYOFFSET;
lpdi->SetProperty(DIPROP_RANGE,&joy_axis_range.diph);
```

Теперь для осей X и Y джойстика задан диапазон от -1024 до 1024. Этот диапазон является произвольным, но лично мне он нравится. Заметим, что здесь использовалась структура данных под названием DIPROPRANGE. Это структура, которую вы устанавливаете по своему усмотрению. Плохо одно: существует миллион способов задания этой структуры, поэтому она действительно вызывает головную боль.

Однако при помощи приведенного шаблона вы можете по крайней мере задать диапазон любой из осей — просто заменить значения полей joy_axis_range.diph.dwObj и joy_axis_range.diph.dwHow соответствующими значениями.

В качестве второго примера задания свойств рассмотрим мертвую зону (или мертвую полосу) осей X и Y. *Мертвая зона* — это некоторая нейтральная область в центре ручки управления. По вашему желанию ручка управления может смещаться немного в сторону от центра, не отправляя при этом никаких значений. Такая ситуация показана на рис. 9.12.

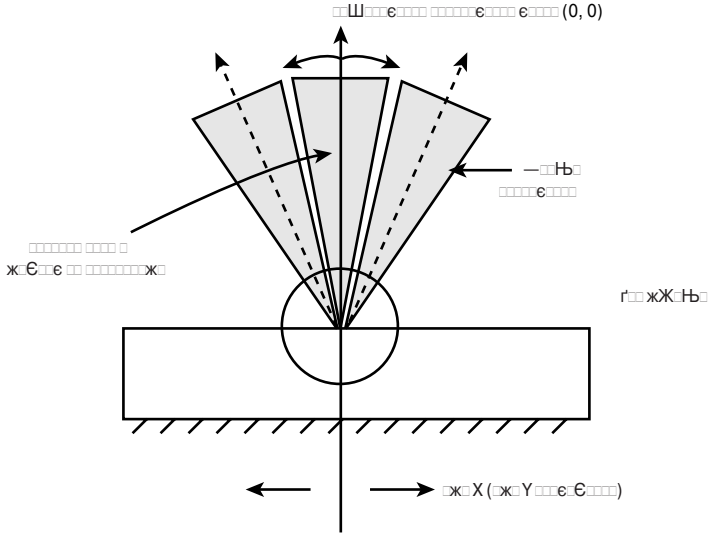


Рис. 9.12. Механика мертвой зоны джойстика

В предыдущем коде, например, вы установили для осей X и Y диапазон от -1024 до 1024, поэтому, если вам захочется иметь для обеих осей мертвую зону размером 10%, нужно задать для нее примерно 102 единицы в обоих направлениях: + и -. Правильно? *Неправильно!!!* Мертвая зона выражается в единицах абсолютного диапазона, составляющего 0–10000, независимо от того, какой диапазон был задан для джойстика. Таким образом, нужно вычислять 10% от 10000 (а не от 1024): $10\% \times 10000 = 1000$. Именно это число вы и должны использовать.

ВНИМАНИЕ

Мертвая зона всегда выражается в единицах от 0 до 10000 (или в сотых долях процентов). Если вам нужна мертвая зона величиной в 50% — используйте число 5000, 10% — число 1000 и т.д.

Поскольку эта операция немного проще, здесь применяется только структура DIPROPWORD.

```
typedef struct _DIPROPWORD
{
  DIPROPHEADER diph;
  DWORD      dwData;
} DIPROPWORD, *LPDIPROPWORD;
```

Эта структура гораздо проще структуры DIPROPRANGE, рассмотренной в предыдущем примере. Теперь выполняем следующие действия:

```
DIPROPWORD dead_band;  // Интересующее нас свойство

dead_band.diph.dwSize = sizeof(dead_band);
dead_band.diph.dwHeaderSize = sizeof(dead_band.diph);
dead_band.diph.dwObj = DIJOFS_X;
dead_band.diph.dwHow = DIPH_BYOFFSET;

// Число 100 используется для обоих направлений диапазона:
dead_band.dwData = 1000;
// Определение свойства
lpdijoy->SetProperty(DIPROP_DEADZONE,&dead_band.diph);
// То же самое для оси Y
dead_band.diph.dwSize = sizeof(dead_band);
dead_band.diph.dwHeaderSize = sizeof(dead_band.diph);
dead_band.diph.dwObj = DIJOFS_Y;
dead_band.diph.dwHow = DIPH_BYO/FFSET;
dead_band.dwData = 1000;
lpdijoy->SetProperty(DIPROP_DEADZONE,&dead_band.diph);
```

Захват джойстика

Теперь попробуйте захватить джойстик посредством вызова функции Acquire().

```
// Захват джойстика
if (FAILED(lpdijoy->Acquire()))
  { /* ошибка */ }
```

По завершении работы с джойстиком не забудьте отменить захват с помощью функции Unacquire(), которая вызывается непосредственно перед обращением к функции интерфейса Release(), освобождающей само устройство.

Опрос джойстика

Джойстики — это единственные устройства, которые требуют опроса (по крайней мере пока). Дело в том, что драйверы некоторых джойстиков генерируют прерывания, и их данные всегда являются "свежими". Другие драйверы менее интеллектуальны (или более эффективны) и должны опрашиваться. Независимо от того, какой точки зрения придерживается разработчик драйвера, перед тем как попытаться прочитать данные, вы *всегда* должны вызвать для джойстика функцию Poll().

```
if (FAILED(lpdijoy->Poll()))
  { /* ошибка */ }
```

Чтение данных о состоянии джойстика

Итак, вы готовы к считыванию данных джойстика (к этому времени вы уже должны быть экспертом в данной области). Выполните обращение к функции GetDeviceState(), которой вы должны передать правильные параметры, исходя из используемого формата

данных — c_dfDIJoystick (c_dfDIJoystick2 для устройств с виброотдачей) — и структуры данных, куда будут помещены полученные данные, — DIJOYSTATE. Вот пример соответствующего кода:

```
DIJOYSTATE joystate; // Для хранения данных джойстика

// ...где-то в основном цикле...

// Считывание состояния джойстика
if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE),
      (LPVOID)joystate)))
  { /* ошибка */ }
```

Теперь у вас есть данные о состоянии джойстика и вы можете работать с ними. Однако не следует забывать, что данные находятся в определенном диапазоне. Давайте напишем небольшую программу, которая перемещает объект почти так же, как и в примере с мышью. И если пользователь нажимает пусковую кнопку (как правило, ее индекс равен 0), срабатывает оружие.

```
// Определение
#define JOYSTICK_FIRE_BUTTON 0

// Глобальные переменные
DIJOYSTATE joystate; // Для хранения данных джойстика

int object_x = SCREEN_CENTER_X, // Изначально размещаем
  object_y = SCREEN_CENTER_Y;   // объект в центре экрана

// ... где-то в основном цикле ...

// Чтение состояния джойстика
if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE),
      (LPVOID)joystate)))
   { /* ошибка */ }

// Перемещение объекта
...

// Состояние кнопки
if (mousestate.rgbButtons[JOYSTICK_FIRE_BUTTON]&0x80)
  { /* Стрельба из оружия */ }
```

Отсоединение джойстика от сервиса

По завершении работы с джойстиком вы, как обычно, должны отменить его захват и освободить устройство. Для этого используется следующий код:

```
// Отменить захват джойстика
if(lpdijoy)
  lpdijoy->Unacquire();

// Освобождение джойстика
if(lpdijoy)
  lpdijoy->Release();
```

ВНИМАНИЕ

Освобождение, выполненное до отмены захвата, может привести к неприятным последствиям. Убедитесь в том, что сначала отменен захват и только *потом* выполняется освобождение.

На прилагаемом компакт-диске находится небольшая демонстрационная программа под названием DEMO9_3.CPP, демонстрирующая работу с джойстиком (рис. 9.13). Как обычно, для ее компиляции в проект необходимо включить библиотеки DDRAW.LIB, DINPUT.LIB и WINMM.LIB, а также библиотеку T3DLIB1.CPP.

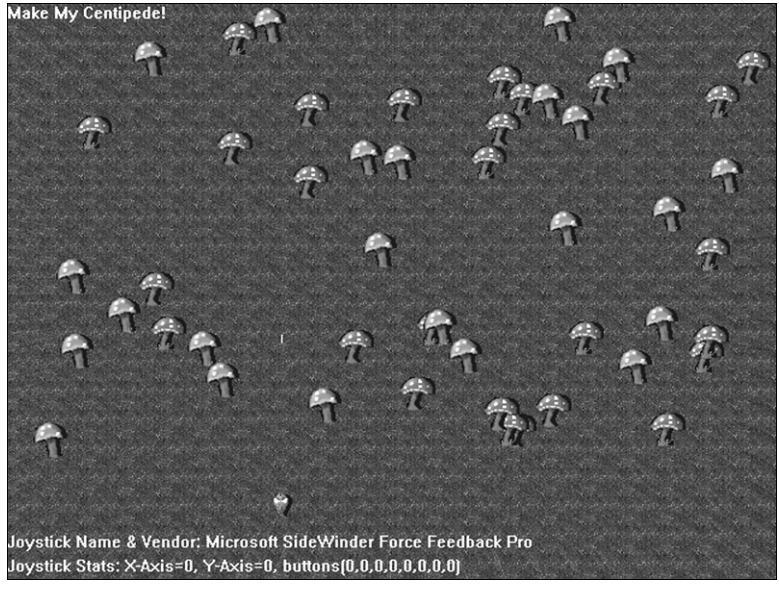


Рис. 9.13. Программа DEMO9_3.EXE в действии

Эффективная обработка данных

Теперь, когда вам известно, как прочитать любое устройство ввода, возникает вопрос, связанный с архитектурой системы ввода. Иными словами, вы можете получать входные данные от нескольких устройств ввода, но иметь при этом отдельный код для каждого устройства — вообщем, сплошная головная боль.

Естественно, появляется мысль о создании общей входной записи, объединяющей все вводимые данные, получаемые от мыши, клавиатуры и джойстика, и последующем использовании этой структуры в работе программы (рис. 9.14).

Пусть, например, вы хотите, чтобы игрок имел возможность использовать все устройства (клавиатуру, мышь и джойстик) одновременно. Например, спусковым крючком оружия может быть левая кнопка мыши и одновременно — клавиша <Ctrl> и первая кнопка джойстика. Кроме того, вам хочется, чтобы все перечисленные далее события заставляли игрока двигаться вправо: игрок перемещает мышь вправо, нажимает на клавиатуре клавишу со стрелкой, направленной вправо, или двигает джойстик вправо.

В качестве примера давайте создадим действительно простую систему, которая принимает от DirectInput запись, содержащую входные данные клавиатуры, джойстика и мыши, а затем объединяет их в одну запись. Когда вы запрашиваете эту запись, вас не интересует, какое именно устройство послужило источником данных: мышь, клавиатура или джойстик, поскольку все входные события будут масштабированы и нормализованы.

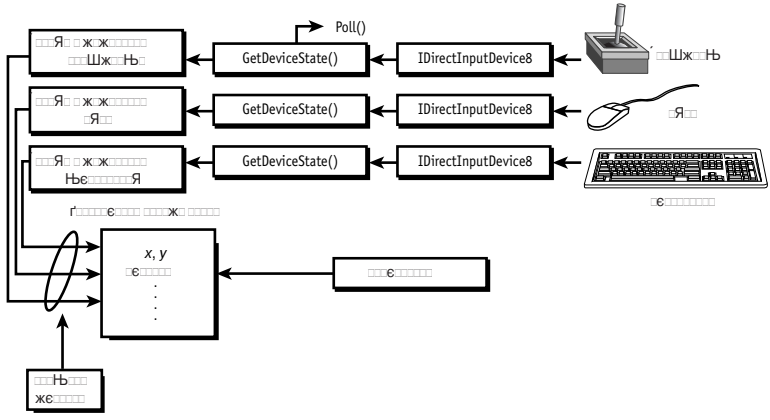


Рис. 9.14. Слияние данных в одну виртуальную запись

Система, которую мы собираемся реализовать, будет иметь следующие характеристики:

- ось X;
- ось Y;
- спусковая кнопка оружия;
- специальная кнопка.

Движение по осям X и Y осуществляется перемещением мыши и джойстика, как обычно (из-за мертвой зоны значения смещения джойстика должны превышать по абсолютной величине 32); на клавиатуре для этого используются клавиши перемещения курсора. Спусковой кнопкой служат: у мыши — левая кнопка, у джойстика — первая кнопка, на клавиатуре — клавиша <Ctrl>; специальной кнопкой — правая кнопка мыши, вторая кнопка джойстика, клавиша <Esc>.

Теперь необходимо создать соответствующую структуру данных для сохранения результата.

```
typedef struct INPUT_EVENT_TYP
{
  int dx;       // Изменение по оси x
  int dy;       // Изменение по оси y
  int fire;     // Спусковая кнопка
  int special;  // Специальная кнопка
} INPUT_EVENT, *INPUT_EVENT_PTR;
```

Далее предположим, что данные из всех устройств ввода прочитаны с помощью примерно такого кода:

```
// Клавиатура
if (FAILED(lpdikey->GetDeviceState(256,
      (LPVOID)keystate)))
  { /* ошибка */ }

// Мышь
if (FAILED(lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE),
      (LPVOID)mousestate)))
  { /* ошибка */ }
```

```
// Джойстик
if (FAILED(lpdijoy->Poll()))
  { /* ошибка */ }

if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE),
      (LPVOID)joystate)))
  { /* ошибка */ }
```

Теперь у нас есть структуры keystate[], mousestate и joystate, готовые к работе. Приведем пример функции, которая могла бы выполнить работу по слиянию данных в одну запись.

```
void Merge_Input (INPUT_EVENT_PTR event_data,  // Результат
      UCHAR       *keydata,                    // Данные клавиатуры
      LPDIMOUSESTATE mousedata,                // Данные мыши
      LPDIJOYSTATE  joydata)                   // Данные джойстика

{
  // Объединение всех данных

  // На всякий случай очищаем запись
  memset(event_data,0,sizeof(INPUT_EVENT));

// Первая спусковая кнопка
if (mousedata->rgbButtons[0]||joydata->rgbButtons[0]||
   keydata[DIK_LCONTROL]) event_data->fire = 1;

// Специальная кнопка
if (mousedata->rgbButtons[1]||joydata->rgbButtons[1]||
   keydata[DIK_ESCAPE]) event_data->special  =  1;

// Движение по оси x вправо
if (mousedata->lX>0 || joydata->lX>32 ||
   keydata[DIK_RIGHT])  event_data->dx = 1;

// Движение по оси x влево
if (mousedata->lX<0 || joydata->lX< —32 ||
   keydata[DIK_LEFT])   event_data->dx = -1;

// Движение по оси y вниз
if (mousedata->lY>0 || joydata->lY>32 ||
   keydata[DIK_DOWN])   event_data->dy = 1;

// Движение по оси y вверх
if (mousedata->lY<0 || joydata->lY<-32 ||
   keydata[DIK_UP])     event_data->dy = -1;

}  // Merge_Data
```

Просто, не правда ли? Конечно, вы можете значительно усложнить этот код, добавляя разные проверки, масштабируя данные и т.п., но главная идея понятна.

Библиотека системы обобщенного ввода T3DLIB2.CPP

Написание простого набора интерфейсных функций для системы DirectInput почти не требует наличия головного мозга. (Ну ладно, у кого для этого не хватает спинного —

что ж...) Большая часть этих функций предельно проста. Все, что требуется сделать, — создать API с очень простым интерфейсом и крайне немногочисленными параметрами, в функции которого входит следующее:

- инициализация системы DirectInput;
- настройка и захват клавиатуры, мыши и джойстика (или любой их разновидности);
- считывание данных из любых устройств ввода;
- закрытие системы, отмена захвата и освобождение всех ресурсов.

Я создал такой API, который находится в файлах T3DLIB2.CPP и T3DLIB2.H на прилагаемом компакт-диске. Этот API делает все необходимое для инициализации системы DirectInput и считывает данные из любого устройства. Однако я не выполнял здесь никакого слияния входных данных, как это было сделано в примере, описанном в одном из предыдущих разделов. Вы будете получать входные данные, представленные в виде структур состояния (состояний) стандартных устройств DirectInput, и обрабатывать различные поля внутри каждой отдельной структуры состояния устройства (клавиатуры, мыши и джойстика). Но при этом вы получаете максимум свободы.

Перед рассмотрением конкретных функций предлагаю взглянуть на рис. 9.15, который показывает взаимосвязь между устройствами и потоком данных.

Вот глобальные переменные этой библиотеки:

```
LPDIRECTINPUT8     lpdi;            // Объект DirectInput
PDIRECTINPUTDEVICE8  lpdikey;       // Клавиатура DirectInput
LPDIRECTINPUTDEVICE8 lpdimouse;  // Мышь DirectInput
LPDIRECTINPUTDEVICE8 lpdijoy;       // Джойстик DirectInput
GUID  joystickGUID;                 // GUID основного джойстика
char  joyname [80];                 // Имя джойстика

// Все вводимые данные сохраняются в этих записях
UCHAR keyboard_state[256];  // таблица состояний клавиатуры
DIMOUSESTATE mouse_state;   // Состояние мыши
DIJOYSTATE joy_state;       // Состояние джойстика
int      joystick_found;    // Отслеживает наличие джойстика
```

Вводимые данные, полученные от клавиатуры, помещаются в массив keyboard_state[], данные мыши сохраняются в mouse_state, а данные джойстика — в joy_state. Эти записи представляют собой стандартные структуры состояний устройств DirectInput. Заметим, что мышь и джойстик считаются в некоторой степени эквивалентными, поскольку предусматривают использование позиции, задаваемой координатами x и y.

Теперь перейдем к функциям. Переменная joystick_found является булевой переменной, которая устанавливается в том случае, когда вы запрашиваете доступ к джойстику. Если джойстик найден, она принимает значение true; в противном случае ее значение false, что позволяет создавать условные операторы при использовании джойстика. Так, без всяких дополнительных хлопот, вы получите новый API.

Прототип функции:

```
int DInput_Init(void);
```

Назначение:

Функция DInput_Init() инициализирует систему ввода DirectInput. Она создает основной COM-объект и в случае успешного завершения возвращает true, а в противном слу-

чае — false. Разумеется, глобальная переменная lpdi получает корректное значение. Эта функция не создает каких-либо устройств. Вот пример инициализации системы ввода:

```
if (!DInput_Init())
  { /*ошибка */ }
```

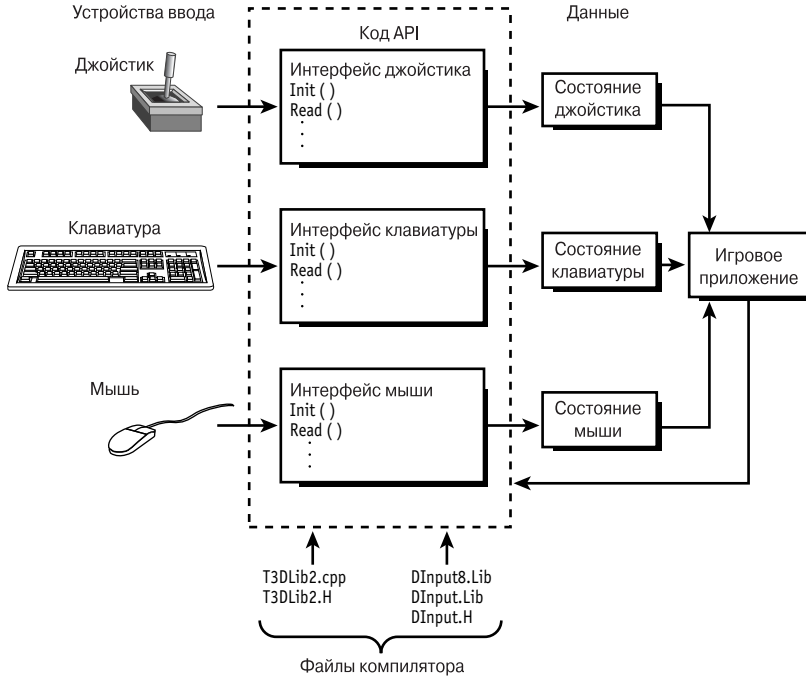


Рис. 9.15. Система программного обеспечения DirectInput

Прототип функции:

```
void DInput_Shutdown(void);
```

Назначение:

Функция DInput_Shutdown() освобождает все COM-объекты и любые ресурсы, выделенные во время обращения к функции DInput_Init(). Обычно вы должны вызывать функцию DInput_Shutdown() в самом конце приложения, после того как освободите все устройств ввода. Вот пример закрытия системы ввода:

```
DInput_Shutdown();
```

Прототип функции:

```
int DInput_Init_Keyboard(void);
```

Назначение:

Функция DInput_Init_Keyboard() инициализирует и захватывает клавиатуру. Она должна работать всегда и возвращать значение true, если никакое другое приложение DirectX не осуществило единоличного захвата клавиатуры. Вот пример использования этой функции:

```
if (!DInput_Init_Keyboard())
  { /* ошибка */ }
```

Прототип функции:

```
int DInput_Init_Mouse(void);
```

Назначение:

Функция DInput_Init_Mouse() инициализирует и захватывает мышь. Она не получает никаких параметров и в случае успешного завершения возвращает true; в противном случае — false. Функция должна успешно работать всегда, если только мышь подключена и не захвачена другим приложением DirectX. При успешном выполнении lpdi становится корректным указателем интерфейса.

```
if (!DInput_Init_Mouse())
  { /* ошибка */ }
```

Прототип функции:

```
int DInput_Init_Joystick(
    int min_x    =-256, // Диапазон по оси x
    int max_x    = 256,
    int min_y    =-256, // Диапазон по оси y
    int max_y    = 256,
    int dead_zone = 10); // Мертвая зона в процентах
```

Назначение:

Функция int DInput_Init_Joystick() инициализирует джойстик. Она принимает пять параметров, которые определяют диапазоны движения X и Y, а также мертвую зону (в процентах). Если вам необходимо использовать значения по умолчанию (от -256 до 256) и мертвую зону, равную 10%, то параметры можно не передавать, так как они имеют значения по умолчанию (специфично для языка C++).

Если функция возвращает true, это означает, что джойстик был найден и настроен, инициализирован и захвачен. После завершения этой функции указатель интерфейса lpdijoy содержит корректное значение и может быть использован для дальнейшей работы. Кроме того, строка joyname[] будет содержать имя устройства джойстика, например Microsoft Sidewinder Pro или нечто подобное.

Вот пример инициализации джойстика и задания его диапазонов от -1024 до 1024, а также 5%-ной мертвой зоны:

```
if (!DInput_Init_Joystick(-1024,1024,-1024,1024,5))
  { /* ошибка */ }
```

Прототипы функции(й):

```
void DInput_Release_Joystick(void);
void DInput_Release_Mouse(void);
void DInput_Release_Keyboard(void);
```

Назначение:

Эти функции освобождают каждое из устройств ввода (джойстик, мышь и клавиатуру) по завершении работы с ними. Они могут быть вызваны даже в том случае, если эти устройства не были инициализированы, поэтому, если хотите, можно просто вызывать их в конце вашего приложения. В приведенном фрагменте кода выполняется инициализация DirectInput, инициализация всех устройств, а затем их освобождение и закрытие.

```
// Инициализация DirectInput
DInput_Init();

// Инициализация и захват всех устройств ввода
```

```
DInput_Init_Joystick();
DInput_Init_Mouse();
DInput_Init_Keyboard();

// ... Выполнение основной работы ...

// Освобождение устройств (порядок значения не имеет)
DInput_Release_Joystick();
DInput_Release_Mouse();
DInput_Release_Keyboard();

// Закрытие DirectInput
DInput_Shutdown();
```

Прототип функции:

```
int DInput_Read_Keyboard(void);
```

Назначение:

Функция DInput_Init_Keyboard() сканирует состояние клавиатуры и помещает данные в массив keyboard_state[] размером 256 байт. Это стандартный массив состояния клавиатуры (в DirectInput), поэтому вы можете использовать при работе с ним символьные константы DIK_*. Если клавиша нажата, значение переменой массива будет равняться 0x80. Вот пример проверки, нажаты ли клавиши перемещения курсора вправо и влево:

```
// Считывание данных клавиатуры
if (!DInput_Read_Keyboard())
  { /* ошибка */ }

// Тестируем данные
if (keyboard_state[DIK_RIGHT])
  { /* переместить объект вправо */ }
else
if (keyboard_state[DIK_RIGHT])
  { /* переместить объект влево */ }
```

Прототип функции:

```
int DInput_Read_Mouse(void);
```

Назначение:

Функция DInput_Init_Mouse() считывает относительное состояние мыши и сохраняет результат в mouse_state — структуре DIMOUSESTATE. Данные соответствуют относительному режиму работы мыши. В большинстве случаев для работы требуются только mouse_starte.lX, mouse_state.lY и rgbButtons[0..2]. Вот пример, описывающий чтение данных мыши и использование ее для перемещения курсора и рисования.

```
// Чтение данных мыши
if (! DInput_Read_Mouse())
  { /* ошибка */ }

// Перемещение курсора
cx + = mouse_state.lX;
cy + = mouse_state.lY;

// проверим, нажата ли левая кнопка
```

```
if (mouse_state.RgbButtons[0])
  Draw_Pixel(cx,cy,col,buffer,pitch);
```

Прототип функции:

```
int DInput_Read_Joystick(void);
```

Назначение:

Функция DInput_Init_Joystick() опрашивает джойстик, а затем считывает его данные в переменную joy_state, которая представляет собой структуру DIJOYSTATE. Если подключенного джойстика нет, функция возвращает значение false и joy_state содержит мусор. Если функция завершается успешно, joy_state содержит информацию о состоянии джойстика. Возвращаемые данные будут принадлежать диапазону, который предварительно был установлен для каждой оси. Значения, описывающие кнопки, являются булевыми величинами, находящимися в массиве rgbButtons[]. Ниже приведен фрагмент кода, использующий джойстик для перемещения объекта вправо и влево; для стрельбы следует воспользоваться первой кнопкой.

```
// Чтение данных джойстика
if (!DInput_Read_Joystick())
  { /* ошибка */ }

// Перемещение объекта
ship_x + = joy_state.lX;
ship_y + = joy_state.lY;

// Проверка кнопки
if (joy_state.RgbButtons[0])
  { /* Стрельба из оружия */ }
```

Конечно, ваш джойстик может иметь свое количество кнопок и несколько осей. В таком случае можно использовать другие поля joy_state согласно их определению в структуре DIJOYSTATE.

Краткий обзор библиотеки T3D

Библиотека T3D состоит их двух основных модулей:

- T3DLIB1.CPP | H — работа с DirectDraw плюс графические алгоритмы;
- T3DLIB2.CPP | H — работа с DirectInput.

Не забывайте о наличии библиотеки при компиляции программ. При необходимости скомпилировать демонстрационную программу DEMOX_Y.CPP посмотрите, какие заголовочные файлы она включает. Если в списке заголовочных файлов содержится какой-либо из библиотечных модулей, необходимо включить в проект соответствующие файлы .CPP.

ВНИМАНИЕ Не забывайте включать в проект библиотеки DDRAW.LIB, DINPUT.LIB и DINPUT8.LIB.

Для демонстрации использования новых библиотечных функций из T3DLIB2.CPP|H я исправил три демонстрационные программы, написанные для этой главы, — DEMO9_1.CPP, DEMO9_2.CPP и DEMO9_3.CPP. Теперь они называются DEMO9_1a.CPP, DEMO9_2a.CPP и DEMO9_3a.CPP соответственно. У вас есть возможность убедиться, какая больш*ая* часть кода может быть выброшена при использовании библиотечных функций.

Не забудьте включить в проект оба исходных библиотечных файла, а также необходимые библиотечные файлы DirectX. И, пожалуйста, не забудьте установить опцию компи-

лятора, обеспечивающую создание .EXE-файла Win32. Я получаю ежедневно десятки писем от людей, которые спрашивают, как настроить компилятор! *Я не оператор технической поддержки компании Microsoft!*

Резюме

Эта глава была довольно интересной, не правда ли? В ней были рассмотрены система DirectInput, клавиатуры, мыши, джойстики, эффективная обработка данных; кроме того, была дополнена новыми функциями библиотека T3D. Вы узнали, что подсистема DirectInput поддерживает все устройств ввода посредством общего интерфейса и для связи с любым устройством нужно просто выполнить несколько подобных операций. Однако трудности, связанные с подсистемами DirectX, на этом не заканчиваются. В следующей главе я перейду к рассмотрению системы DirectSound и немного коснусь DirectMusic. Только после этого вы сможете приступить к серьезному программированию своих игр.

ГЛАВА 10

DirectSound и DirectMusic

Создание звука и музыки на компьютере всегда было сущим кошмаром. Однако с приходом подсистем DirectSound и DirectMusic решение этой проблемы значительно упростилось. В данной главе рассматриваются следующие темы.

- Основы звука
- Цифровой и синтезированный звук
- Аппаратное обеспечение звука
- API подсистемы DirectSound
- Форматы звуковых файлов
- API подсистемы DirectMusic
- Поддержка звука в библиотеке T3D

Программирование звука на компьютере

Как правило, программирование звука всегда оставляют напоследок. Написание звуковой системы связано с большими трудностями не только потому, что это требует глубокого изучения принципов создания звука и музыки, но и потому, что вы должны быть уверены — данная система будет работать с любой звуковой картой. Проблема состоит именно в этом. В прошлом большинство программистов, занимающихся написанием игр, использовали звуковые библиотеки сторонних производителей, например Miles Sound System, DiamondwareSound Toolkit или какую-то другую. Каждая система имеет свои за и против, но самой большой проблемой является цена такой библиотеки. Звуковая библиотека, которая работает под управлением и DOS и Windows, может стоить тысячи долларов.

Беспокоиться по поводу DOS вам больше не нужно, чего нельзя сказать о Windows. Операционная система Windows действительно имеет поддержку звука и мультимедиа, но она никогда не ставила перед собой цель продемонстрировать сверхвысокие рабочие показатели, которые необходимы для видеоигр реального времени. К счастью, подсистемы DirectSound и DirectMusic решают все эти проблемы (а также ряд других). DirectSound и DirectMusic не только распространяются бесплатно, но и обладают чрезвычайно высокими рабочими показателями, могут поддерживать множество различных звуковых карт и имеют расширения, позволяющие делать именно то, что вам нужно.

DirectSound, например, поддерживает трехмерное пространственное звучание (под управлением DirectSound3D), а DirectMusic способна делать намного больше, чем просто воспроизводить MIDI-файлы (Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов). DirectMusic представляет собой новую технологию реального времени, предназначенную для сочинения и воспроизведения музыки и основанную на DLS-данных (Downloadable Sounds — загружаемые звуки). Это означает не только то, что музыка будет одинаково звучать на любой звуковой карте, но и то, что система DirectMusic может "на ходу" создавать музыку для вашей игры; в основе этой характеристики лежат *шаблоны*, *лейтмотивы* и *индивидуальные особенности*, которые вы программируете заранее. Использование искусственного интеллекта подсистемы DirectMusic для сочинения музыки потребует от вас значительных усилий, но может оказаться весьма эффективным для тех игр, в которых нужно изменять тональность музыки в зависимости от хода игры (вы же не хотите самостоятельно сочинять 10–20 различных версий каждой песни?). И конечно, в DirectX 8.0 реализована более тесная интеграция DirectSound и DirectMusic под управлением DirectX Audio. В более ранних версиях (до версии 8.0) эти интерфейсы были наполовину самостоятельными и связь между ними можно было реализовать с трудом. А теперь давайте немного поговорим о звуке.

А затем был звук...

Звук относится к таким физическим явлениям, которые имеют циклическое определение. Если выйти на улицу и спросить людей, что такое звук, большинство, вероятно, ответили бы так: "Гм... это такая штука, которую вы слышите ушами, она похожа на звуки и шумы". (Не верите? Идите попробуйте...) И это действительно так, но подобное определение никак не проясняет реальной физики звука, которую стоит знать, если вы собираетесь записывать, обрабатывать и воспроизводить звук.

Как видно из рис. 10.1, звук представляет собой волну механического давления, излучаемую некоторым источником. Звук может существовать только в некоторой среде, например атмосфере, заполненной такими газами, как азот, кислород гелий и т.д. Звук может распространяться и в воде, но там он движется намного быстрее, чем в воздухе, поскольку повышенная плотность среды увеличивает ее проводимость.

На самом деле звуковая волна представляет собой перемещение молекул. При работе динамика окружающий воздух слегка перемещается вперед-назад, создавая области разрежения и повышенного давления, которые образуют волну, достигающую ваших ушей. Поскольку звук передается благодаря распространению волны в воздухе посредством механических столкновений молекул, ему нужно для этого какое-то время. Именно поэтому звук распространяется медленно (вообще говоря, относительно медленно). Вы можете увидеть, как что-то случилось, например авария автомобиля, и ничего при этом не слышать в течение одной-двух секунд, если место происшествия находится достаточно далеко. Это объясняется тем, что в воздухе механическая (или звуковая) волна способна распространяться только со скоростью около 344 m/s; эта скорость в большей или меньшей

степени зависит от плотности и температуры воздуха. В табл. 10.1 представлены значения скорости звука в воздухе, морской воде и стали (для средних значений температуры).

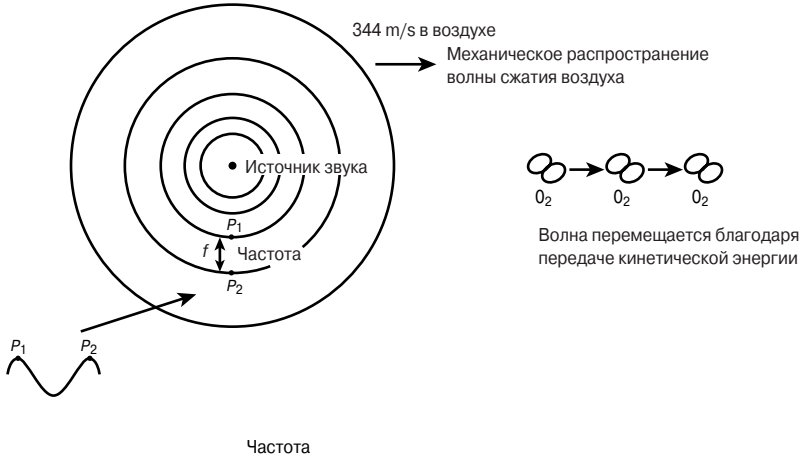


Рис. 10.1. Звуковая волна

Таблица 10.1. Скорость звука в различных материалах

Материал/Среда	*Примерная скорость звука, m/s*
Воздух	344
Морская вода	1476
Сталь	5064

Взглянув на табл. 10.1, вы сразу поймете, почему гидролокатор так хорошо работает под водой и не способен работать в воздухе (одна из причин — слишком медленная скорость звука в воздухе). Импульс гидролокатора (ультразвуковой импульс) распространяется под водой со скоростью 1478 m/s. Если вы сравните это значение со средней скоростью звука в воздухе, вам сразу станет ясно, почему гидролокатор почти мгновенно сканирует объекты, движущиеся под водой на приемлемом расстоянии.

$\int\sum_{\alpha}^{\infty}$

Если вам интересно, скорость звука равна произведению частоты звука на длину волны, т.е. $f\lambda$. Кроме того, скорость звука в среде может быть вычислена как $c=\sqrt{K/\rho}$, где K — модуль объемной упругости, а ρ — плотность невозмущенной среды. Для изотропной твердой среды модуль объемной упругости следует заменить модулем сдвига.

Итак, звук — это механическая волна, которая распространяется в воздухе с постоянной скоростью, а именно скоростью звука. Распространяющаяся звуковая волна может иметь два параметра: амплитуду и частоту. Амплитуда определяет объем перемещаемого воздуха. Неутомимый оратор перемещает больше воздуха, поэтому его звуки более громкие. Частота звука определяет, сколько полных волн или циклов в секунду излучается источником. Она измеряется в герцах (Hz). Человек в среднем способен слышать звуки, обладающие частотой в диапазоне 20–20 000 Hz.

Мужчина в среднем обладает голосом в диапазоне от 20 до 2000 Hz, в то время как голос женщина имеет диапазон от 70 до 3000 Hz. Мужчины имеют более низкие голоса, женщины — более высокие.

На рис. 10.2 представлены амплитуда и частота, характерные для некоторых стандартных типов звуковых волн.

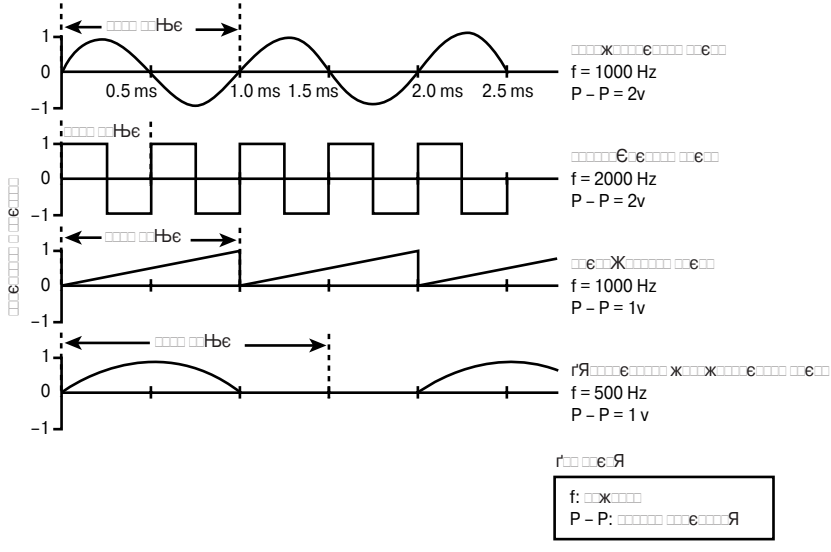


Рис. 10.2. Различные формы волн

Форму волны можно рассматривать как форму изменений амплитуды звука. Даже если два звука имеют одинаковую амплитуду и частоту, из-за различной формы волны они будут звучать для нас по-разному.

И наконец, мы слышим звуки при помощи ушей, которые могут показаться довольно простыми, но это далеко не так. Наши уши имеют множество маленьких *ресничек*, каждая из которых может обнаруживать волны определенного частотного диапазона. Когда звук попадает в ухо, по мере того как волна передает импульсы давления, эти реснички вибрируют и резонируют, в зависимости от конкретных звуков, и посылают сигналы в мозг, который затем преобразует эти сигналы в сознательное восприятие звука. Однако существа, живущие на других планетах, вполне возможно, могут "видеть" звук, поэтому помните, что понятие звука является чисто субъективным. Неизменными остаются лишь распространение и физика звука. Однако это справедливо только для тех регионов Вселенной, где нет искривления пространства, которое характерно, например, для окрестности черной дыры или автострад в Калифорнии.

Я повторю, что звук — это волна давления, которая перемещает воздух. Скорость смены растяжений и сжатий в волне называется *частотой*, а количество перемещаемого воздуха определяется относительной *амплитудой*, или громкостью звука. Кроме того, звук может характеризоваться формой волны: синусоидальные волны, прямоугольные волны, пилообразные волны и т.п. — все они имеют разную форму волны. Люди могут слышать звуки в диапазоне частот от 20 до 20 000 Hz, а средняя частота человеческого голоса составляет примерно 2000 Hz. Однако это еще не все.

Чистый тон всегда имеет вид синусоидальной волны и может обладать любой частотой и амплитудой. Одиночные тона можно услышать в электронных игрушках или в те-

лефонах с тональным набором (технически в тональных аппаратах предусмотрены два тона на одну кнопку (стандарт DTMF), но они очень близки по частоте). В реальном же мире подавляющее большинство звуков, например голоса, музыка и окружающие шумы (скрип дверей или ремонт за стеной), состоят из сотен или даже тысяч чистых тонов, которые перемешаны друг с другом. А следовательно, звук имеет спектр.

$\int\sum_{\alpha}^{\infty}$

Основной формой волны является синусоидальная, описываемая функцией $\sin \omega t$. Все остальные типы волн могут быть представлены посредством линейной комбинации нескольких синусоидальных волн с помощью преобразования Фурье, что можно доказать математически.

Спектр звука представляет собой его частотное распределение. На рис. 10.3 представлено частотное распределение моего голоса. Очевидно, что мой голос имеет множество различных частот, но большинство из них — низкие. Главное, что вы должны понимать для создания реальных звуков: звуки состоят из множества одиночных простых тонов, имеющих разные частоты и амплитуды.

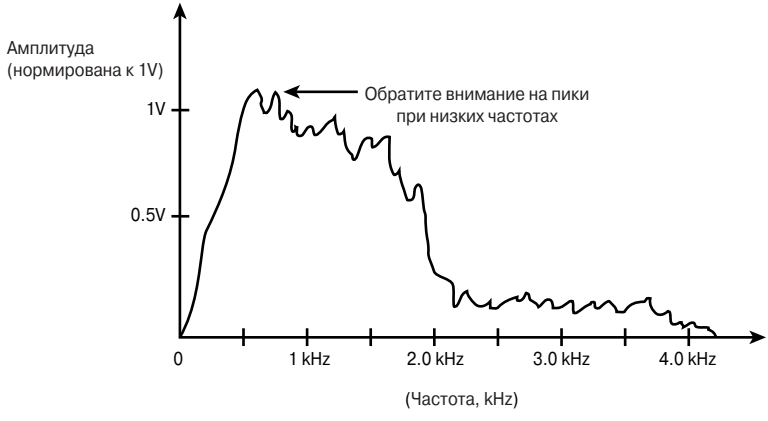


Рис. 10.3. Частотный спектр обычного мужского голоса

Все это замечательно, но ваша цель — заставить компьютер издавать звуки. Нет проблем: компьютер может управлять динамиком при помощи электрических сигналов, заставляя его перемещать воздух назад и вперед с любой скоростью, применяя при этом любую силу (в пределах разумного). Осталось узнать, как это делается.

Цифровой звук и MIDI

Компьютер способен издавать звуки двух видов: *цифровые* и *синтезированные*. Цифровой звук представляет собой заранее сделанную запись, тогда как синтезированные являются программируемым воспроизведением звуков, основанным на алгоритмах и аппаратных звуковых генераторах. Цифровые звуки обычно использовались для звуковых эффектов, например взрывов или разговоров людей, в то время как синтезированные чаще применялись при создании музыки. И в настоящее время наблюдается та же картина: синтезированные звуки чаще всего применяются только для музыкальных, но не для звуковых эффектов. Впрочем, в свое время, в частности в 80-х годах, программисты, писавшие игры, использовали синтезаторы с частотной модуляцией для имитации звуков двигателей, взрывов, ружейных выстрелов, стуков, сирен и т.д. Будучи более-менее удов-

летворительными, они все же не звучали так же хорошо, как цифровые звуковые эффекты, поэтому впоследствии их перестали использовать.

Цифровой звук

Цифровой звук предусматривает выполнение *оцифровывания*, которое означает преобразование аналоговых данных в цифровую форму, т.е. в нули и единицы — 110101010110. В то время как электрический сигнал может создавать звуки, заставляя магнитное поле перемещать конический магнит динамика, говорящий в динамик создает обратный эффект: динамик генерирует электрический сигнал в зависимости от того, какие колебания он воспринимает. Электрический сигнал несет в себе информацию о звуке, закодированную в виде аналогового электрического сигнала (рис. 10.4).

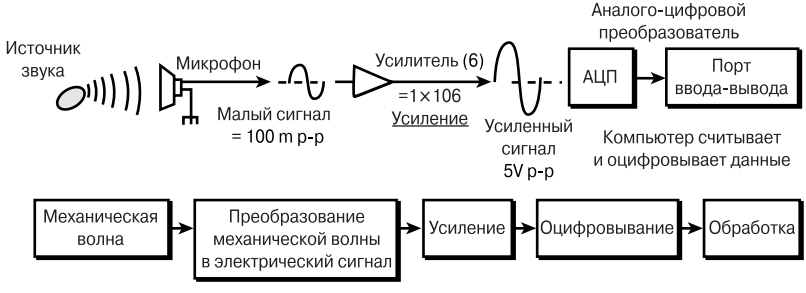


Рис. 10.4. Преобразование звука

С помощью соответствующих устройств это напряжение, содержащее закодированную информацию о звуке, может быть преобразовано в цифровую форму. Именно так работает ваш плеер для компакт-дисков. Информация, содержащаяся на компакт-дисках, имеет цифровую форму, тогда как информация на лентах — аналоговую. Цифровая информация намного легче поддается обработке, и это единственная информация, которую могут обрабатывать цифровые компьютеры. Поэтому, чтобы компьютер смог обработать звук, его нужно преобразовать в поток цифровых данных при помощи аналого-цифрового преобразователя, осуществляющего, как следует из названия, преобразование сигнала из аналоговой формы в цифровую. Этот процесс показан на рис. 10.5,*а*.

После того как звук записан в память компьютера, он может быть обработан или воспроизведен при помощи преобразователя, выполняющего обратное преобразование из цифровой формы в аналоговую (см. рис. 10.5,*б*). Главное — прежде чем работать со звуковой информацией, ее нужно преобразовать в цифровой формат. Но записать цифровой звук не так просто, так как звук несет в себе очень много информации. Если вы хотите записать реальный звук, то должны учесть два фактора: частоту и амплитуду.

Количество выборок в секунду (выполненных вами при записи звука) называется частотой выборки. Для точного воспроизведения звука она должна по крайней мере в два раза превышать частоту исходного звука. Иными словами, осуществляя запись человеческого голоса, диапазон частот которого составляет от 20 до 2000 Hz, вы должны производить выборку звука с частотой не ниже 4000 Hz.

Все это имеет математическое обоснование, основанное на том, что все звуки состоят из синусоидальных волн. Если вы можете прочитать синусоидальную волну с наивысшей частотой, то сможете прочитать все волны с более низкими частотами, которые образуют этот звук. Но, чтобы прочитать синусоидальную волну с частотой f, вы должны читать ее с частотой $2f$. Если скорость считывания будет равна f, вы не сможете определить, где находитесь — в нижней или верхней точке волны (в пределах одного цикла). Иными слова-

ми, для воспроизведения любой синусоидальной волны необходимо иметь две точки. Если вас заинтересует эта тема, поинтересуйтесь теоремами Котельникова и Шеннона, а также частотой Найквиста.

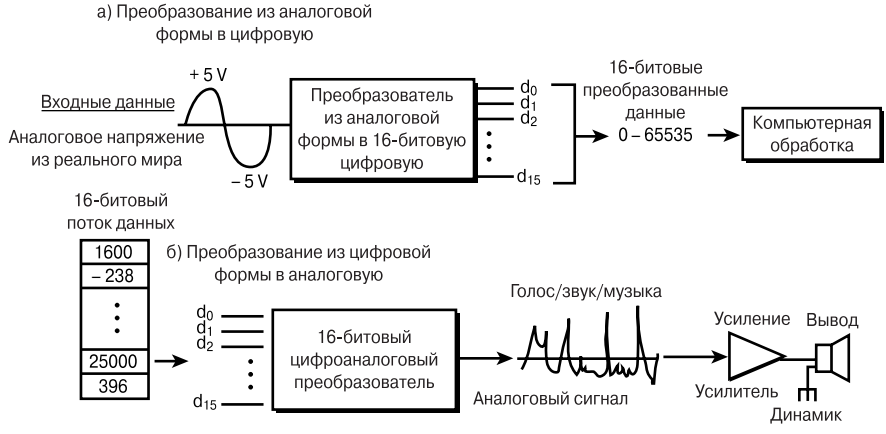


Рис. 10.5. Шестнадцатибитовое аналого-цифровое и цифроаналоговое преобразование

Вторым параметром выборки является *разрешение по амплитуде*, т.е. число различных значений амплитуды, которое вы в состоянии записать. Если вы располагаете только восемью битами на одну выборку, это означает, что существует только 256 возможных значений амплитуды. Этого достаточно для игр, но для воспроизведения профессиональных звуков и музыки нужно иметь по крайней мере 16-битовое разрешение, допускающее 65 536 различных значений.

Итак, по своей сути цифровой звук — это запись и/или считывание звука, преобразованного в цифровую форму из аналогового сигнала. Цифровой звук хорошо применим для непродолжительных звуковых эффектов, но плохо подходит для продолжительного звучания, поскольку его потребность в памяти достаточно велика: выборка 16 бит, с частотой 44.1 kHz требует примерно 88 Кбайт в секунду (конечно, если ваша игра поставляется на компакт-диске, то нет проблем выделить пару сотен мегабайт для чисто цифровой музыки). И наконец, в 99 случаях из 100 цифровой звук намного лучше, чем синтезированный (хотя под управлением DirectMusic синтезированная музыка звучит почти так же хорошо, как и цифровая).

Синтезированный звук и MIDI

Хотя наилучшим звучанием в настоящее время обладает цифровой звук, синтезированный в силу долгого использования также становится все лучше и лучше. Синтезированный звук не записывается в цифровом виде; это просто математическое воспроизведение звука, основанное на его описании. Исходя из описания желаемого звука, синтезаторы используют аппаратное обеспечение и алгоритмы для генерации звуков “на ходу”. Допустим, вам хочется услышать чистую ноту “ля” частотой 440 Hz. В таком случае можно спроектировать некоторое аппаратное обеспечение, которое генерирует чистую аналоговую синусоидальную волну любой частоты из диапазона от 0 до 20 000 Hz, а затем дать указание создать тон частотой 440 Hz. Это и есть основа синтеза.

Единственная проблема заключается в том, что многие хотят слышать не один тон, а одновременно несколько (конечно, если они не отдают предпочтение музыкальным по-

здравительным открыткам), поэтому нам необходимы технические средства, которые поддерживают по крайней мере 16–32 различных тона одновременно (рис. 10.6). Различные видеоигры использовали нечто подобное еще в 70-х и 80-х годах, но особого восторга они не вызывали. Дело в том, что в состав большинства звуков входит множество частот: звуки имеют полутона, обертоны и гармоники (т.е. частоты, кратные основной частоте). Именно они и придают звукам своеобразие и насыщенность.

ВНИМАНИЕ

Обычно при описании звуков я не использую такие слова, как *своеобразный* и *насыщенный*, но здесь я должен это сделать, поскольку эти термины широко используются музыкальными критиками. Так что отнеситесь к этому с пониманием.

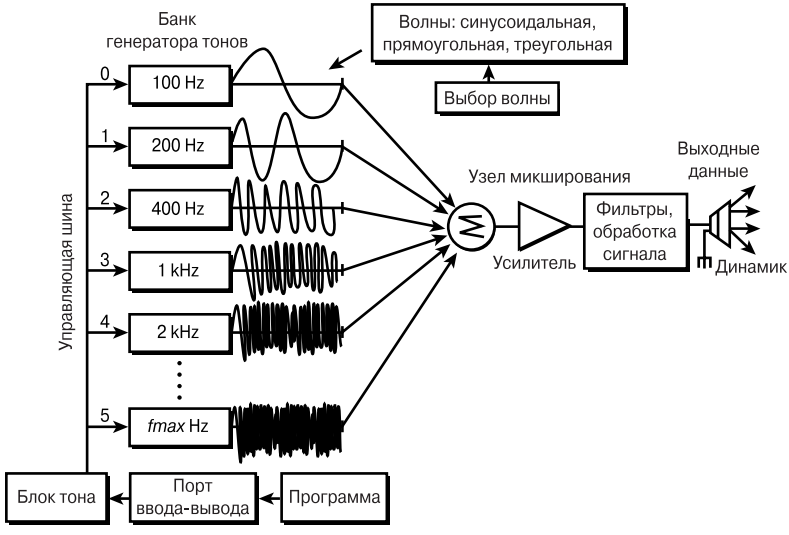


Рис. 10.6. Синтез звука с использованием каналов

Первой попыткой получить улучшенный звук можно считать *синтез с частотной модуляцией* (Frequency Modulation — FM). Помните старую карту Ad-Lib? Она была предшественницей карты Sound Blaster и первой компьютерной платой, предназначенной для поддержки многоканального синтеза с частотной модуляцией. FM-синтезатор может изменять не только амплитуду синусоидальной звуковой волны, но и частоту этой волны.

Математическую основу FM-синтеза определяет использование обратной связи. Вывод FM-синтезатора возвращается на вход и таким образом осуществляется модуляция сигналов, а также создание гармоник и тонов с фазовым сдвигом. Практический результат состоит в том, что по сравнению с одиночными тонами они звучат весьма реалистично.

MIDI

Примерно в то же самое время, когда появились все эти штучки для FM-синтеза, появился и новый формат файлов, предназначенный для синтеза музыки, который получил название MIDI (Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов). MIDI представляет собой язык, который описывает музыкальные композиции как функцию от времени. Вместо оцифровывания звука MIDI описывает его при помощи ключей, инструментов и специальных кодов. MIDI-файл, например, может выглядеть следующим образом:

```
Тихое воспроизведение ноты "фа" на канале 1
Громкое воспроизведение ноты "ми" на канале 2
Выключить канал 1
.
.
.
Выключить все каналы
```

Конечно, вся эта информация закодирована и представлена в виде двоичного последовательного потока данных, но вы имеете представление о картине в целом. Каждый канал в MIDI-описании связан с отдельным инструментом или звуком. У вас может быть 16 каналов, каждый из которых представляет определенный инструмент, например фортепиано, барабаны, гитару, контрабас, флейту, трубу и т.д. Поэтому MIDI — это косвенный метод кодирования музыки.

Однако он предусматривает выполнение синтеза аппаратным обеспечением и записывает только реальные музыкальные ноты и временны*е* интервалы. Увы, но на одном компьютере MIDI-музыка может звучать совершенно по-иному, чем на другом, что объясняется способом выполнения синтеза звучания инструментов. С другой стороны, MIDI-файл, обеспечивающий звучание музыки в течение одного часа, может занимать всего лишь несколько сотен килобайт памяти (сравните это с мегабайтами памяти, необходимыми для той же музыки в цифровом виде). Такая экономия порой может иметь большое значение.

Единственная проблема, связанная с технологиями MIDI и FM-синтеза, заключается в том, что они хороши только для музыки. Конечно, вы можете спроектировать FM-синтезаторы, предназначенные для создания продолжительного шума взрывов или лазерных вспышек, но эти звуки всегда будут простыми и не такими естественными, как цифровые. Поэтому в дальнейшем были созданы более совершенные методы аппаратного синтеза, например технологии табличного синтеза или волноводов.

Аппаратное звуковое обеспечение

В настоящее время существует три основных типа синтеза звука: FM, табличный (а также его программные версии) и волноводный. FM-синтез мы уже рассмотрели, теперь вкратце рассмотрим два других типа.

Табличный синтез

Табличный синтез представляет собой нечто среднее между синтезом и цифровой записью. Он работает примерно так. Таблица волн содержит ряд реальных оцифрованных звуков. Затем эти данные обрабатываются при помощи DSP (Digital Signal Processor — процессор цифровых сигналов), который получает реальный образец и воспроизводит его по вашему желанию с любой частотой и амплитудой. Это значит, что вы можете отобрать образец реальной игры на фортепьяно, а затем воспроизвести любую ноту на этом фортепьяно, используя табличный синтез. Звучание получается почти таким же хорошим, как и цифровой, но вам нужно иметь оцифрованный исходный звук, что опять же требует затрат памяти. Хорошим примером использования такой технологии можно считать Creative Labs AWE32.

В дополнение к аппаратной таблице волн существуют программные табличные системы, например формат MOD, предназначенный для Amigas, и система DLS, используемая в DirectMusic. Современные компьютеры имеют такие высокие скорости работы, что если у вас есть просто преобразователь из цифровой формы в аналоговую, который по-

зволяет воспроизводить цифровые звуки, то вы можете применять его для синтеза звука, программно используя образцы звука реальных инструментов, что очень похоже на табличный синтез. До тех пор пока вы сможете заставлять DSP работать в режиме реального времени и выполнять обработку частот, амплитуд и другие похожие операции, вам не нужно никакого другого аппаратного обеспечения! Именно так и работает DirectMusic.

Волноводный синтез

Это самая последняя технология синтеза. Благодаря применению микросхем цифровой обработки сигналов и весьма специализированному аппаратному обеспечению синтезатор звука действительно может виртуально генерировать математическую модель инструмента, а затем просто воспроизводить его звук. Это может показаться научной фантастикой, тем не менее это факт. При использовании данной технологии человеческое ухо не в состоянии отличить звук такого инструмента от реального, так что вы можете создавать MIDI-файлы, которые будут управлять табличным или волноводным синтезатором, и получать при этом великолепные результаты. Такую технологию применяет Creative Labs AWE64 Gold и другие более мощные системы.

Итак, наш вердикт таков. Синтезатор способен создавать абсолютно реальную музыку, при этом сама музыка может быть закодирована в формате MIDI. Если же вам нужно использовать речь или специальные звуковые эффекты, с помощью синтезаторов они могут быть реализованы с большим трудом и даже в случае применения волноводной технологии вам потребуется специальное программное обеспечение.

Используя DirectMusic, можно запрограммировать воспроизведение инструментов с помощью цифровых звуков, так что данная проблема решаема. Цифровой звук можно применять для всех звуковых эффектов, а DirectMusic — для музыки. Да, здесь потребуется немного больше работы, чем для простого воспроизведения волнового файла, но зато DirectMusic одинаково звучит на *всех* машинах, является бесплатной, может читать стандартные MIDI-файлы и обладает массой полезных характеристик, которые могут понадобиться вам при работе. Поэтому логично прийти к решению об использованию обеих подсистем: DirectSound для звука и DirectMusic для музыки.

Цифровая запись: инструменты и технологии

Прежде чем завершить вводный экскурс в область звука и музыки, мне хочется дать вам несколько советов по поводу записи звука и музыки для ваших игр, поскольку я получаю по электронной почте множество писем на эту тему. Существует, как минимум, три способа создания цифровых образцов.

- Получить их из реального мира с помощью микрофона или другого ввода данных.
- Приобрести образцы звуков в цифровом или аналоговом формате и загрузить или записать их для последующего использования.
- Синтезировать цифровые звуки с помощью синтезатора типа Sound Forge.

Третий метод несколько старомоден, но он может оказаться полезным в том случае, если вам нужно создать чистые тона при помощи цифрового оборудования и у вас нет источника звука, который можно было бы записать. Но самыми важными для нас являются первые два метода.

Если вы создаете игру, в которой много речи, вам, вероятно, потребуется записать собственный голос (или голоса друзей). “Отшлифуйте” его при помощи соответствующего про-

граммного обеспечения, а затем используйте в игре. В играх, которые применяют стандартные звуки взрывов, хлопающих дверей, грохота и т.п., вы, вероятно, сможете выйти из положения, используя универсальные аудиоклипы. Например, почти каждый, имеющий отношение к этому бизнесу, обладает собственной копией библиотеки звуков Sound Ideas General 6000/7000+. Существует примерно 40 компакт-дисков, содержащих тысячи звуковых эффектов, используемых в фильмах, поэтому здесь есть все. Но когда в фильме я слышу звук открываемой двери из игр DOOM/Quake, хочется просто заткнуть свои бедные уши!

Единственная проблема, связанная с использованием библиотек, — их высокая стоимость: примерно 2500 долларов за приличный лицензионный продукт. Так что же теперь делать? В любом компьютерном магазине можно найти пятидолларовые компакт-диски, содержащие звуковые эффекты. Вы могли бы приобрести несколько таких дисков; двух-трех вполне достаточно для работы (там вы найдете образцы звуков машин, космических кораблей, чудовищ и т.д.). Но поскольку я не только автор этой книги, но еще и хороший парень, то намерен предоставить в ваше распоряжение полный набор звуков, которые использовал в одной из своих игр. Они находятся на прилагаемом компакт-диске в каталоге `SOUND\`. Все они имеют формат `.WAY` и `.MID`, так что вы вполне можете использовать их в своих играх. Но не исключено, что вы захотите несколько их изменить и подправить.

Запись звуков

Если вы записываете свои собственные звуки, я рекомендую следующие настройки: ведите 16-битовую запись в режиме 22 kHz монозвука. Помните, *никакого стерео*. Система DirectSound наилучшим образом работает при использовании монозвука, поэтому запись в режиме стерео не даст никакого эффекта, а кроме того, большинство звуков, которые вы создаете или записываете, в любом случае будут монофоническими, поэтому их запись в режиме стерео просто потребует ненужных затрат памяти.

Если вы записываете с микрофона, подключенного к вашей звуковой карте, купите хороший микрофон. Кроме того, производите запись в закрытой комнате без всякого фонового шума или помех. Если вы записываете непосредственно с некоторого устройства, например проигрывателя компакт-дисков или радио, убедитесь в надежности соединений и используйте только высококачественные соединители.

И наконец, используйте для ваших звуковых файлов подходящие имена. Никаких тайн: вы никогда не вспомните, что есть что, если не упорядочите информацию. И не забывайте, что на дворе XXI век, — присваивайте файлам длинные имена.

Обработка звуков

Если вы уже оцифровали звуки с помощью Sound Forge или аналогичного программного обеспечения, то, вероятно, захотите выполнить последующую их обработку. И вновь Sound Forge или аналогичный продукт может помочь вам сделать всю необходимую обработку. Вы сможете обнаружить любые дефекты, отрегулировать громкость, устранить шумы, добавить эхо и т.д. Однако на время выполнения подобных операций рекомендую сделать резервные копии звуков, чтобы не запортить оригиналы. Присвойте обработанным звукам новые имена, добавив в конце цифры или что-нибудь еще.

Во время обработки звука поэкспериментируйте с преобразованием частот, искажениями, эхо и другими эффектами. Если вы обнаружили красивый эффект, обязательно запишите его *формулу*, чтобы иметь возможность воспроизвести его. Не могу даже сказать, сколько раз я добивался совершенного женского компьютерного голоса (полученного посредством обработки моего собственного), а потом забывал, как я это делал.

И наконец, когда вы завершите свою работу со звуками, перепишите их все в одном и том же формате, например в режиме моно, 22 или 11 kHz и 8 или 16 бит. Этим вы очень

поможете DirectSound при обработке звуков. Если у вас есть звуки с различной частотой оцифровывания и разным количеством бит на образец, DirectSound всегда будет приводить их к своему стандарту — 22 kHz и 8 бит.

СОВЕТ

С технической точки зрения внутренний формат DirectSound — это режим 8-битового стерео, 22 kHz. Однако большинство звуков являются по своей природе монофоническими, и передача данных DirectSound в режиме стерео требует ненужных затрат, если только вы действительно не осуществляете запись при помощи двух микрофонов, помещенных в разные места, или не имеете настоящих стереоданных.

DirectSound

DirectSound, как и DirectDraw, состоит из ряда компонентов или интерфейсов. Однако данная книга посвящена программированию игр, поэтому мы кратко остановимся только на самых важных из них. Из этого следует, что я не намерен рассматривать компонент DirectX, предназначенный для реализации трехмерного пространственного звучания, DirectSound3D или интерфейс захвата звука DirectSoundCapture. Я собираюсь уделить внимание только основным интерфейсам системы DirectSound. Поверьте мне, этого будет вполне достаточно, чтобы сполна обеспечить вас работой.

На рис. 10.7 показаны взаимосвязи между DirectSound и другими подсистемами Windows. Заметим, что эта схема очень похожа на соответствующую схему для DirectDraw. Однако DirectSound имеет одну важную характеристику, которой нет у DirectDraw: если у вас нет драйвера DirectSound для вашей звуковой карты, DirectSound все равно будет работать, но вместо драйвера она будет использовать эмуляцию и интерфейс драйвера устройства операционной системы Windows. До тех пор пока вы будете распространять свой продукт вместе с .DLL-файлами DirectSound, ваш код будет работать даже в том случае, если у пользователя нет драйверов DirectSound для его карты. Пусть не так быстро, но будет!

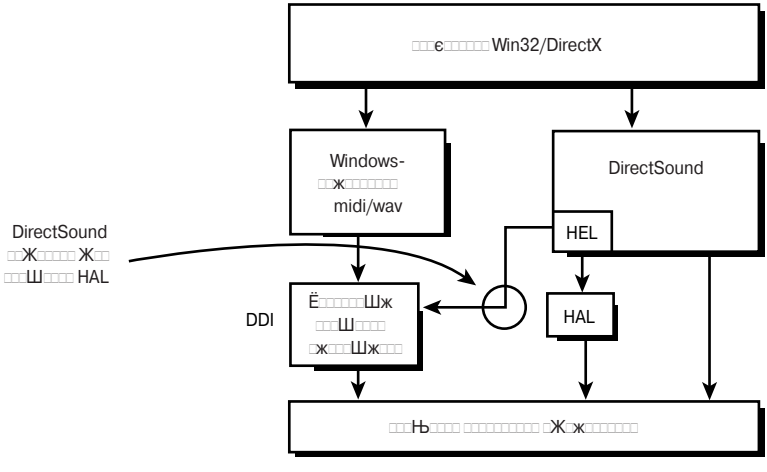


Рис. 10.7. DirectSound и Windows

Рассмотрим компоненты DirectSound:

- библиотека .DLL, которая загружается в случае использования DirectSound;
- компилируемая библиотека и заголовочный файл — DSOUND.LIB и DSOUND.H соответственно.

Единственное, что вам нужно для создания приложения DirectSound, — это включить указанные файлы в свой проект, и все будет прекрасно.

Для использования DirectSound вы должны создать COM-объект DirectSound, а затем запросить различные интерфейсы из этого основного объекта. На рис. 10.8 представлены основные интерфейсы системы DirectSound.

- Iunknown. COM-объект, который является базовым для всех остальных COM-объектов.
- IDirectSound. Основной COM-объект системы DirectSound. Он представляет непосредственно звуковое аппаратное обеспечение. Если на вашем компьютере есть одна или несколько звуковых карт, необходимо создать объект DirectSound для каждой из них.
- IDirectSoundBuffer. Представляет аппаратные средства смешения и реальные звуки. Существуют звуковые буферы DirectSound двух видов: *основные* и *дополнительные* (вы не находите аналогию между DirectSound и DirectDraw?). Имеется только один основной буфер, и он представляет звук, воспроизводимый в данный момент и смешиваемый при помощи аппаратных (будем надеяться) или программных средств. Дополнительные буферы представляют сохраняемые и подлежащие воспроизведению звуки. Они могут находиться в системной памяти или звуковой оперативной памяти (Sound RAM — SRAM) на звуковой плате. В любом случае вы можете воспроизводить из вспомогательного буфера столько звуков, сколько хотите, и до тех пор, пока у вас хватает мощности и памяти делать это. На рис. 10.9 представлена взаимосвязь между основным и вспомогательными звуковыми буферами.
- IDirectSoundCapture. Этот интерфейс применяться не будет, но мне хотелось бы заметить, что он используется для записи и перехвата звуков. Применение этого интерфейса позволяет игроку записать свое имя, или если вы большой технофанат, то можете с его помощью перехватывать речь в реальном времени для распознавания голоса.
- IDirectSoundNotify. Этот интерфейс используется для передачи сообщений системе DirectSound. Необходимость в этом может возникнуть в том случае, если игра имеет сложную звуковую систему, но можно обойтись и без него.

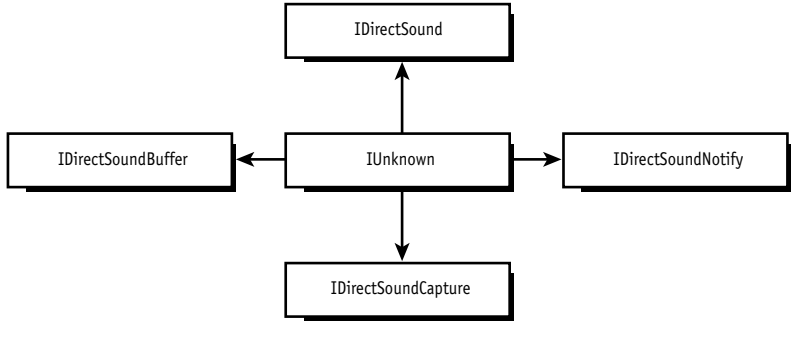


Рис. 10.8. Интерфейсы DirectSound

Для использования DirectSound вначале нужно создать основной объект DirectSound, создать один или несколько звуковых буферов, загрузить в них звуки, а затем воспроизвести любой звук, который вам нужен. DirectSound позаботится обо всех деталях, включая смешение звуков. Итак, давайте начнем с создания непосредственно самого объекта DirectSound.

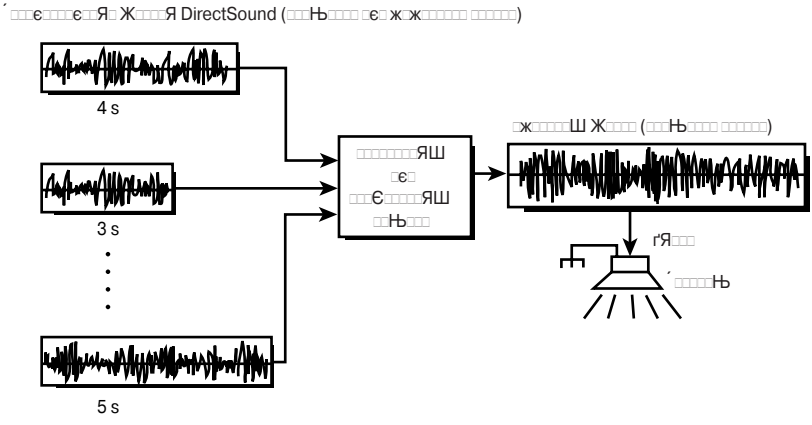


Рис. 10.9. Звуковые буферы

НА ЗАМЕТКУ

DirectX 8.0 имеет новый интерфейс DirectSound — IDirectSound8, так что Microsoft благополучно пропустила версии 2–7. Однако это ничего не меняет в нашем рассмотрении, поскольку мы собираемся использовать только стандартные интерфейсы DirectSound, которые были доступны, начиная с DirectX 3.0.

Запуск системы DirectSound

Основной объект DirectSound представляет звуковую карту (карты). Если у вас несколько звуковых карт, вы должны их перечислить, найти и запросить их GUID. Но, если вы хотите подсоединиться к звуковому устройству, которое задано по умолчанию, достаточно просто создать объект DirectSound, который будет представлять основную звуковую карту. Вот указатель на интерфейс, который представляет объект DirectSound:

```
LPDIRECTSOUND lpds; // указатель на интерфейс DirectSound
```

Для создания объекта DirectSound вы должны обратиться к функции DirectSoundCreate(), прототип которой выглядит так:

```
HRESULT DirectSoundCreate(
  LPGUID lpGuid,              // GUID звуковой карты, NULL для
                              // устройства по умолчанию
  LPDIRECTSOUND *lpDS,        // указатель на интерфейс объекта
  IUnknown FAR *pUnkOuter) // всегда NULL
```

Обращение к этой функции очень похоже на вызов, который используется для создания основного объекта системы DirectDraw. В целом все они выглядят одинаково: если вы освоили одну часть системы DirectX, то легко справитесь со всеми остальными. Проблема в том, что Microsoft добавляет новые интерфейсы примерно с той же скоростью, с какой вы в состоянии их изучать... Чтобы создать объект DirectSound, нужно сделать следующее:

```
LPDIRECTSOUND lpds; // указатель на объект DirectSound

// Создать объект DirectSound
if (DirectSoundCreate(NULL, &lpds, NULL) != DS_OK)
  { /* ошибка */ }
```

Заметим, что теперь успешное завершение обозначается как DS_OK, а не DD_OK. Однако надежнее проверять успешность/неудачу работы функции так же, как и ранее — с помощью макросов FAILURE()и SUCCESS().

```
// создать объект DirectSound
if (FAILED(DirectSoundCreate(NULL, &lpds, NULL)))
  { /* ошибка */ }
```

И конечно, по завершении работы с объектом DirectSound вы должны освободить его с помощью вызова

```
lpds->Release();
```

Эта операция выполняется на стадии завершения вашего приложения.

Понятие уровня взаимодействия

После того как вы создали основной объект DirectSound, самое время задать уровень взаимодействия. DirectSound несколько сложнее, чем DirectDraw, поскольку в ней имеется такая вещь, как уровень взаимодействия.

Для DirectSound можно задать несколько уровней взаимодействия. Они подразделяются на две группы: настройки, позволяющие контролировать основной аудиобуфер, и настройки, которые не позволяют этого делать. Помните, основной аудиобуфер представляет реальные аппаратные (или программные) средства смешивания, которые выводят звук в динамик, так что, если вы ошибетесь при работе с ним, это может привести к сбоям и искажению звуков не только вашего приложения, но и других. Приведем общее краткое описание каждого уровня взаимодействия.

- **Обычный уровень взаимодействия (Normal Cooperation)**. Самый "взаимодействующий" среди всех остальных уровней. Пока ваше приложение имеет фокус ввода, оно может воспроизводить звуки, но то же справедливо и для других приложений. Кроме того, вы не имеете права на запись данных в основной аудиобуфер, и DirectSound создаст для вас основной буфер по умолчанию: 22 kHz, стерео, 8 бит. Я считаю, что именно этот уровень должен использоваться в подавляющем большинстве случаев.
- **Приоритетный уровень взаимодействия (Priority Cooperation)**. При задании этого уровня вы получаете преимущественный доступ ко всему аппаратному обеспечению; вы можете изменять настройки основного микшера или затребовать звуковые аппаратные средства для выполнения дополнительных операций с памятью, например сжатия. Установка этого уровня необходима только в том случае, если вам требуется изменять формат данных в основном буфере, что может потребоваться, например, при воспроизведении 16-битового звука.
- **Исключительный уровень взаимодействия (Exclusive Cooperation)**. То же, что и приоритетный уровень, но ваше приложение будет слышно только во время работы на переднем плане.
- **Уровень взаимодействия Write_Primary (Write_Primary Cooperation)**. Это уровень наивысшего приоритета. Вы имеете полный контроль над звуком и должны сами управлять основным буфером. Этот режим используется только в том случае, если вы пишете собственный звуковой микшер или подсистему.

Задание уровня взаимодействия

На мой взгляд, вы должны использовать обычный уровень приоритета. Это самый простой способ работы и без лишних проблем. Чтобы задать уровень взаимодействия, нужно вызвать функцию SetCooperativeLevel() интерфейса основного объекта DirectSound. Вот ее прототип.

```
HRESULT SetCooperativeLevel(
      HWND hwnd,     // Дескриптор окна
      DWORD dwlevel); // Уровень взаимодействия
```

Функция возвращает DS_OK в случае успешного завершения, в противном случае — некоторое иное значение. Необходимо обязательно выполнить проверку на наличие ошибок, поскольку более чем вероятно, что другое приложение приняло на себя управление звуковой картой. В табл. 10.2 представлены значения флагов, задающих различные уровни взаимодействия.

Таблица 10.2. Значения для уровней взаимодействия DirectSound

Значение	*Описание*
DSSCL_NORMAL	Задает обычный уровень взаимодействия
DSSCL_PRIORITY	Задает приоритетный уровень взаимодействия, позволяя устанавливать формат данных основного аудиобуфера
DSSCL_EXCLUSIVE	Задает приоритетный уровень взаимодействия в дополнение к исключительному доступу при работе на переднем плане
DSSCL_WRITEPRIMARY	Предоставляет полный контроль над основным аудиобуфером

Итак, обычный уровень взаимодействия после создания объекта DirectSound задается следующим образом:

```
if (FAILED(lpds->SetCooperativeLevel(main_window_handle,
                    DSSCL_NORMAL)))
  { /* Ошибка при задании уровня взаимодействия */ }
```

Просто, не правда ли? Рассмотрите демонстрационную программу DEMO10_1.CPP на прилагаемом компакт-диске, которая создает объект DirectSound, задает уровень взаимодействия, а затем освобождает объект. Пока она не производит никаких звуков, это наша следующая задача.

СОВЕТ При компиляции демонстрационных программ в ходе чтения данной главы убедитесь в том, что не забыли включить в проект библиотеку DSOUND.LIB.

Основной и дополнительный аудиобуферы

Объект DirectSound, который непосредственно представляет звуковую карту, имеет один основной буфер. Основной буфер определяет аппаратные (или программные) средства микширования и выполняет непрерывную обработку данных, что можно представить себе в виде маленькой конвейерной ленты. Ручное управление буфером — дело весьма сложное, но вы, к счастью, не должны этим заниматься. До тех пор пока вы не зададите для уровня взаимодействия наивысший приоритет, об основном буфере будет заботиться DirectSound. Кроме того, вам не придется создавать основной буфер, поскольку DirectSound сама создаст его для вас (если, конечно, для уровня взаимодействия будет задан не наивысший приоритет, а, например, DSSCL_NORMAL).

Единственный недостаток при работе через DirectSound заключается в том, что основной буфер имеет следующие характеристики: 22 kHz, 8-битовый стереозвук. Если вам нужен 16-битовый звук или более высокая частота, придется задать уровень взаимодействия, как минимум, DSSCL_PRIOPITY, а затем определить новый формат данных для основного буфера. Но сейчас просто воспользуемся значениями по умолчанию, поскольку это намного облегчит нашу жизнь.

Работа с дополнительными буферами

Дополнительные, или вторичные, буферы представляют реальные звуки, которые должны быть воспроизведены. Эти буферы могут иметь любые размеры, которые вы задаете по своему усмотрению (если, конечно, для этого хватает памяти). Однако количество данных, которые может хранить оперативная память на звуковой карте, ограничено, поэтому следует соблюдать осторожность, если вы хотите хранить звуки непосредственно в самой звуковой карте. Но звуки, сохраняемые непосредственно в звуковой карте, требуют существенно меньшей обработки при воспроизведении.

В настоящее время используются дополнительные буферы двух видов: *статические* и *потоковые*. Статические аудиобуферы содержат звуки, которые вы должны хранить и воспроизводить снова и снова, и хорошо подходят для хранения в звуковой или системной памяти. Потоковые аудиобуферы несколько отличаются от статических. Представьте себе, что вы хотите воспроизвести весь компакт-диск при помощи DirectSound. Я не думаю, что вы располагаете достаточным количеством системной или звуковой оперативной памяти для сохранения всех 650 Мбайт аудиоданных, поэтому вы должны считывать данные порциями и направлять их поток в буфер DirectSound. Именно для этого и предназначен потоковый буфер. Вы постоянно пополняете его новыми звуковыми данными по мере их воспроизведения. Сложно и непонятно? Давайте рассмотрим рис.10.10.

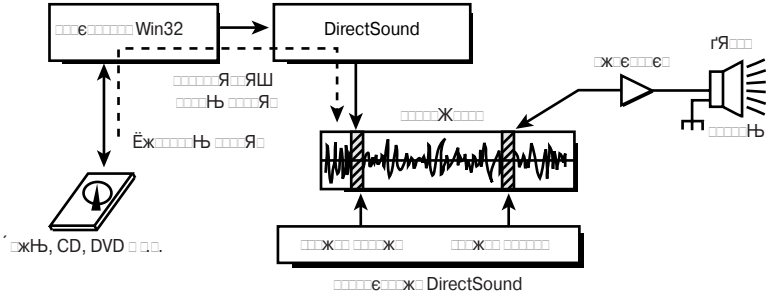


Рис. 10.10. Потоковые аудиоданные

Все дополнительные аудиобуферы могут быть записаны как статические или потоковые. Однако возможна ситуация, когда вы пытаетесь выполнить запись в момент воспроизведения звука, и тогда DirectSound применяет кольцевую буферизацию. Она подразумевает, что каждый звук сохраняется в кольцевом массиве данных, непрерывно считываемых в одном месте с использованием курсора воспроизведения, а записываемых в другом месте (оно находится позади места считывания) с помощью курсора записи. Конечно, если нет необходимости производить запись в аудиобуферы в тот момент, когда они воспроизводят звуки, то не стоит беспокоиться по этому поводу, но вам не избежать этого, если аудиоданные направляются в виде потока.

При работе с потоком ввод новых данных должен быть синхронизирован с их чтением, поскольку в силу кольцеобразности буфера при опережающей записи можно переполнить буфер и начать вести запись на место воспроизводимого звука. Однако для большинства игр все это остается только теорией, поскольку если вы храните в памяти звуковые эффекты продолжительностью до нескольких секунд, а музыкальные дорожки загружаете по запросу, то вам понадобится всего несколько мегабайт оперативной памяти. Выделение 2–4 Мбайт памяти для звука в машине, располагающей памятью свыше 32 Мбайт, нельзя считать слишком большой проблемой.

Создание дополнительных аудиобуферов

Чтобы создать дополнительный аудиобуфер, вам нужно вызывать функцию CreateSoundBuffer(), передавая ей соответствующие параметры. В случае успешного завершения функция создаст аудиобуфер, проинициализирует его и возвратит указатель на его интерфейс:

```
LPDIRECTSOUNDBUFFER lpdsbuffer; // Буфер DirectSound
```

Однако, перед тем как вызвать функцию CreateSoundBuffer(), вы должны настроить описывающую буфер структуру DirectSoundBuffer, которая аналогична описанию DirectDrawSurface. Структура описания имеет тип DSBUFFERDESC и выглядит следующим образом:

```
typedef struct _DSBUFFERDESC
{
DWORD       dwSize;             // Размер структуры
DWORD       dwFlags;            // Управляющие флаги
DWORD       dwBufferBytes;      // Размер аудиобуфера в байтах
DWORD       dwReserved;         // Не используется
LPWAVEFORMATEX lpwfxFormat; // Формат волны
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Поле dwSize определяет размер стандартной структуры DirectX, dwBufferBytes задает размер буфера в байтах, который вы задаете по своему усмотрению, а поле dwReserved не используется. Действительно интересными являются только поля dwFlags и lpwfxFormat. Поле dwFlags содержит значения флагов, используемых при создании аудиобуфера (их краткий список представлен в табл. 10.3).

Таблица 10.3. Флаги, используемые при создании дополнительного буфера

Значение	*Описание*
DSBCAPS_CTRLALL	Буфер должен иметь все средства управления
DSBCAPS_CTRLDEFAULT	Буфер должен иметь средства управления, задаваемые по умолчанию, что подразумевает установку следующих флажков: DSBCAPS_CTRLPAN, DSBCAPS_CTRLVOLUME и DSBCAPS_CTRLFREQUENCY
DSBCAPS_CTRLFREQUENCY	Буфер должен иметь средства управления частотой
DSBCAPS_CTRLPAN	Буфер должен иметь средства управления балансом
DSBCAPS_CTRLVOLUME	Буфер должен иметь средства управления громкостью
DSBCAPS_STATIC	Указывает на то, что буфер будет использоваться для статических аудиоданных. Почти всегда вы будете создавать такие буферы в памяти аппаратных средств (если имеется такая возможность)
DSBCAPS_LOCHARDWARE	Использовать для этого аудиобуфера аппаратное смешение и память (если есть свободная память)
DSBCAPS_LOCSOFTWARE	Заставляет сохранять буфер в программной памяти и использовать программное смешение, даже если установлен флажок DSBCAPS_STATIC и доступны аппаратные ресурсы
DSBCAPS_PRIMARYBUFFER	Указывает на то, что буфер представляет собой основной аудиобуфер. Этот флажок устанавливается только в том случае, если вы хотите создать основной буфер и берете на себя ответственность за все происходящее со звуком

В большинстве случаев используются флаги

DSBCAPS_CTRLDEFAULT | DSBNCAPS_STATIC | DSBCAPS_LOCSOFTWARE

Эти флаги означают задание средств управления по умолчанию, статического звука и системной памяти соответственно. Если вам захочется использовать память аппаратных средств, используйте вместо DSBCAPS_LOCSOFTWARE флаг DSBCAPS_LOCHARDWARE.

НА ЗАМЕТКУ

Чем больше возможностей вывода звука вы задаете, тем большее число ограничений (программных фильтров) он должен пройти перед тем, как будет выведен на динамики. А следовательно, при этом возрастает время обработки. Так что, если вам не нужны средства управления громкостью, балансом и возможность сдвига частоты, забудьте о флаге DSBCAPS_CTRLDEFAULT и воспользуйтесь только теми средствами, которые вам абсолютно необходимы.

Теперь рассмотрим структуру WAVEFORMATEX. Она содержит описание звука, который должен быть представлен в буфере (кстати, это стандартная структура Win32). В эту структуру записываются следующие параметры: скорость воспроизведения, количество каналов (1 — моно или 2 — стерео), количество бит при выборке и т.д.

```
typedef struct _WAVWFORMATEX
{
  WORD  wFormatTag;        // Всегда WAVE_FORMAT_PCM
  WORD  nChannels;         // Число аудиоканалов (1 или 2)
  DWORD nSamplesPerSec;  // Образцов в секунду
  DWORD nAvgBytesPerSec;// Средняя скорость данных
  WORD  nBlockAlign;       // Количество каналов, умноженное на
                           // количество байт в выборке
  WORD  wBitsPerSample;   // Число бит в выборке
  WORD  cbSize;            // В этой книге всегда 0
} WAVWFORMATEX;
```

Все достаточно просто. WAVEFORMATEX содержит описание звука, которое является частью DSBUFFERDESC. Давайте посмотрим, как это делается, и начнем с прототипа функции CreateSoundBuffer().

```
HRESULT CreateSoundBuffer (
  LPCDSBUFFERDESC     lpcDSBuffDesc,
                // Указатель на DSBUFFERDESC
  LPLPDIRECTSOUNDBUFFER lplpDSBuff,
                // Указатель на аудиобуфер
  IUnknown FAR       *pUnkOuter);  // Всегда NUUL
```

Теперь приведем пример создания дополнительного буфера DirectSound, имеющего следующие параметры: 8-битовый монозвук, 11 kHz, область памяти достаточна для хранения двух секунд звука.

```
// Указатель на DirectSound
LPDIRECTSOUNDBUFFER lpdsbuffer;
DSBUFFERDESC      dsbd;    // Описание буфера DirectSound
WAVEFORMATEX      pcmwf; // Описание формата

// Задание структуры данных формата
emset(&pcmwf,0,sizeof(WAVEFORMATEX));
pcmwf.wFormatTag     = WAVE_FORMAT_PCM;
pcmwf.nChannels      = 1;          // Число каналов = 1
pcmwf.nSaamplesPerSec = 11025; // Частота оцифровки 11 kHz
pcmwf.nBlockAlign    = 1;          // См. ниже
```

```
// Задаем количество данных в блоке.
// В нашем случае 1 канал, умноженный на 1 байт на выборку
// (всего 1 байт; в случае стереофонии — 2 байта)
pcmwf.nAvgBytesPerSec =
    pcmwf .nSamplesPerSec * pcmwf .nBlockAlign;
pcmwf.wBitsPerSample = 8; // 8 бит на образец
pcmwf.cbSize = 0;              // всегда 0

// Описание буфера DirectSound
memset(dsbd,0,sizeof(DSBUFFERDESC));
dsbd.dwSize  = sizeof(DSBUFFERDESC);
dsbd.dwFlags = DSBCAPS_CTRLDEFAULT|DSBCAPS_STATIC|
         DSBCAPS_LOCSOFTWARE;
dsbd.dwBuffertBytes = 22050; // Достаточно для 2 секунд при
                             // частоте 11025
dsbd.lpwfxFormat = &pcmwf;   // структура WAVEFORMATEX

// Создаем буфер
if (FAILED(lpds->CreateSoundBuffer(&dsbd,&lpdsbuffer,NULL)))
  { /* ошибка */ }
```

Если функция завершилась успешно, создается новый аудиобуфер и указатель на него передается в переменную lpdsbuffer. Буфер готов к воспроизведению; проблема только в том, что в нем ничего нет! Вы сами должны заполнить аудиобуфер данными. Это можно сделать, считывая данные из звукового файла, сохраненные в формате .VOC, .WAV, .AU или в каком-то другом, затем разобрав их и заполнив буфер. Вы можете также создать алгоритмические данные и самостоятельно записать их в буфер в качестве теста. Давайте посмотрим, каким образом можно записать данные в буфер, а о том, как прочитать звуковые файлы, которые находятся на диске, вы узнаете несколько позже.

Запись данных во вторичные буферы

Как уже отмечалось, дополнительные аудиобуферы являются по своей природе кольцевыми, а потому производить в них запись немного сложнее, чем в стандартные линейные массивы данных. Например, когда дело касается поверхностей DirectDraw, вы просто блокируете память поверхности и записываете в нее данные. (Это возможно только потому, что здесь работает драйвер, который делает нелинейную память линейной.) DirectSound поступает аналогично: вы блокируете память, но, вместо того чтобы получить в ответ один указатель, вы получаете два! Поэтому данные в буфер записываются не просто так, а в соответствии со значениями полученных указателей. Чтобы понять сказанное, рассмотрим прототип функции Lock().

```
HRESULT Lock(
  DWORD  dwWriteCursor,      // Позиция курсора записи
  DWORD  dwWriteBytes,       // Блокируемый размер
  LPVOID lplpvAudioPtr1,     // Указатель на первую часть данных
  LPWORD lpdwAudioBytes1, // Размер первой части данных
  LPVOID lplpvAudioPtr2,     // Указатель на вторую часть данных
  LPWORD lpdwAudioBytes2, // Размер второй части данных
  DWORD  dwFlags );          // Способ блокирования
```

Если вы задаете в параметре dwFlags значение DSBLOCK_FROMWRITECURSOR, буфер будет заблокирован, начиная от текущей позиции курсора записи этого буфера. Если вы зададите в dwFlags значение DSBLOCK_ENTIREBUFFER, будет заблокирован весь буфер.

Допустим, например, что вы создаете аудиобуфер, размеры которого позволяют хранить 1000 байт. Если вы заблокируете буфер для записи, то в ответ получите два указателя, а также длины сегментов памяти, выделенные для записи. Первый сегмент может иметь длину, равную 900 байт, а второй — 100 байт. Суть в том, что вы должны записать первые 900 байт в первую область памяти, а вторые 100 байт — во вторую область памяти. Рис. 10.11 поясняет, как это делается.

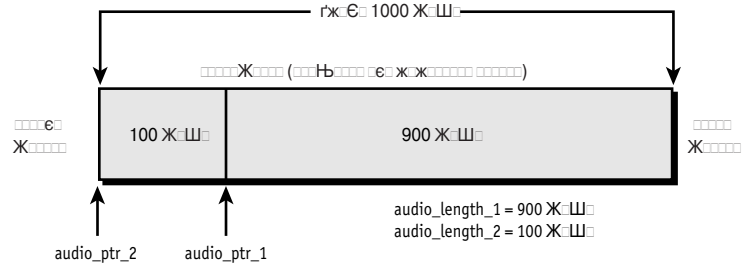


Рис. 10.11. Блокировка аудиобуфера

Приведем пример блокировки аудиобуфера размером 1000 байт.

```
UCHAR *audio_ptr_1,     // Используется для получения
    *audio_ptr_2;       // указателей буфера
int   audio_length_1,   // Длина каждого сегмента буфера
    audio_length_2;

// Заблокировать буфер
if (FAILED(lpdsbuffer->Lock(0,1000,
        (void**)&audio_ptr_1, &audio_length_1,
        (void**)&audio_ptr_2, &audio_length_2,
        DSBLOCK_ENTIREBUFFER))
  { /* ошибка  */ }
```

Заблокировав буфер, вы можете записывать данные в память. Эти данные могут быть получены из файла или же созданы при помощи некоторого алгоритма. По окончании записи в аудиобуфер необходимо разблокировать его при помощи вызова функции Unlock(), которая принимает в качестве параметров оба указателя и оба значения длины.

```
if (FAILED(lpdsbuffer->Unlock(audio_ptr_1, audio_length_1,
                  audio_ptr_2, audio_length_2)))
   { /* Проблема при разблокировании */ }
```

И как всегда, завершив работу с аудиобуфером, вы должны удалить его при помощи вызова Release():

```
lpdsbuffer->Release();
```

Нельзя удалять аудиобуфер, если он может еще когда-нибудь понадобиться, так как тогда вам придется загружать его снова.

Теперь давайте посмотрим, каким образом можно воспроизводить звуки с помощью DirectSound.

Воспроизведение звука

Если вы создали все необходимые аудиобуферы и заполнили их звуками, значит, все готово к их воспроизведению (конечно, если хотите, звуки можно создавать и удалять "на ходу"). DirectSound имеет ряд управляющих функций, предназначенных для воспроизведения звуков и изменения параметров воспроизведения. Допускается изменение громкости, частоты, стереобаланса и т.п.

Управление звуком

Чтобы воспроизвести звуки, находящиеся в аудиобуфере, применяется функция Play().

```
HRESULT Play(
  DWORD dwReserved1,
  DWORD dwReserved2, // Оба параметра - 0
  DWORD dwFlags);      // Управляющие флаги воспроизведения
```

Единственный определенный в DirectSound флаг — это DSBPLAY_LOOPING. Указание этого флага приводит к циклическому исполнению звука. Если вам нужно воспроизвести его один раз, задайте в dwFlags значение, равное 0. Вот пример, в котором звук воспроизводится снова и снова:

```
if (FAILED(lpdsbuffer->Play(0,0,DSBPLAY_LOOPING)))
  { /* Ошибка */ }
```

Циклическое выполнение можно использовать, например, для постоянно звучащей фоновой музыки.

Останов звука

Если вы запустили звук, его можно остановить еще до того, как закончится воспроизведение. Чтобы это сделать, применяется функция Stop():

```
HRESULT Stop(); // Все предельно просто
```

Теперь покажем, как можно остановить звук, который был запущен в предыдущем примере:

```
if (FAILED(lpdsbuffer->Stop()))
  { /* Ошибка */ }
```

Теперь вы обладаете информацией, достаточной для того, чтобы разобраться с демонстрационной программой DEMO10_2.CPP, которая находится на прилагаемом компакт-диске. В ней создается объект DirectSound и единственный дополнительный аудиобуфер, который затем заполняется синтезированной синусоидальной волной и воспроизводит ее. Эта программа проста, но она эффективно демонстрирует все, что вам нужно знать для воспроизведения звука.

Громкость

DirectSound позволяет управлять громкостью, или амплитудой звука, однако не совершенно свободно. Если ваши аппаратные средства не поддерживают изменение громкости, DirectSound будет заново микшировать звук, используя для этого новое значение амплитуды, что может потребовать дополнительных вычислительных затрат. В любом случае прототип функции для изменения громкости выглядит так:

```
HRESULT SetVolume(LONG lVolume); // Ослабление в децибелах
```

Функция SetVolume() работает не так, как можно было бы предполагать. Вместо того чтобы давать системе DirectSound указание насчет увеличения или уменьшения амплитуды, SetVolume() управляет ослаблением (или, если хотите, антиусилением) звука. Если вы передаете функции значение 0 (что эквивалентно DSBVOLUME_MAX), звук будет воспроизводиться без ослабления, т.е. с максимальной громкостью. Значение -10000 (DSBVOLUME_MIN) приводит к максимальному ослаблению, равному -100 dB, и тогда вы ничего не услышите.

Самое лучшее, что можно сделать в данном случае, — создать для этой операции функцию-оболочку с тем, чтобы можно было использовать значение от 0 до 100 или что-нибудь в этом роде, более понятное и подходящее. Такую работу выполняет, например, следующий макрос:

```
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100—volume)))
```

Теперь громкость задается в диапазоне от 0 до 100, где 100 — это полная громкость, а 0 — полная тишина. Напишем пример воспроизведения звука, громкость которого составляет 50%[1] от максимальной величины:

```
if (FAILED(lpdsbuffer->SetVolume(DSVOLUME_TO_DB(50))))
  { /* Ошибка */ }
```

НА ЗАМЕТКУ

Если вы не знаете, что такое децибел, то могу сказать, что это просто мера звука или мощности, основанная на единице бел, названной в честь изобретателя телефона Александра Грэхэма Белла (Alexander Graham Bell). Многие вещи в электронике измеряются с помощью *логарифмической* шкалы, и шкала децибел являет собой один из таких примеров. Иными словами, 0 dB означает отсутствие поглощения, -1 dB означает, что звук составляет 1/10 от своего первоначального значения, -2 dB — 1/100 и т.д. Поэтому, если звук характеризуется ослаблением -100 dB, его просто невозможно услышать.

Заметим, что ряд шкал использует другое значение множителя для децибела — это может быть как другая степень 10, так и степень, например, 2. Так, в некоторой шкале -10 dB могли бы означать 1/10, а -20 — 1/100. Это как раз тот случай, когда в разных ситуациях единица измерения может означать совершенно разные вещи...

Изменение частоты

Одним из самых интересных эффектов, которые можно использовать при воспроизведении звука, является изменение частоты его воспроизведения. При этом изменяется основной тон звука, и вы можете сделать его замедленным и злым или быстрым и радостным. Вы можете сделать свой собственный голос похожим на голос бурундука или Дарта Вадера (герой фильма "Звездные войны". — *Прим. перев.*) в реальном времени! Чтобы изменить частоту воспроизведения, воспользуйтесь функцией SetFrequency(), показанной ниже.

```
HRESULT SetFrequency(
  DWORD dwFrequency); // новая частота в диапазоне
                      // от 100 до 100000 Hz
```

А вот как ускорить воспроизведение звука:

```
if (FAILED(lpdsbuffer->SetFrequency(22050)))
  { /* Ошибка */ }
```

[1] Вообще говоря, ослабление в 300 dB, выбранное автором в качестве полной тишины, достаточно условно. Вы можете поэкспериментировать с использованным в макросе значением -30, изменяя его и прослушивая, как будет изменяться звук, ослабленный "на 50%". — *Прим. ред.*

Если исходный звук был оцифрован с частотой 11025 Hz (11 kHz), то новый звук будет воспроизводиться в два раза быстрее, иметь удвоенный основной тон и длительность его воспроизведения уменьшится в два раза.

Настройка стереобаланса

Следующее, что можно сделать со звуком, — изменить стереобаланс, или соотношение мощности, выходящей из каждого динамика. Например, если вы воспроизводите звук при одном и том же значении громкости в обоих динамиках (или наушниках), будет казаться, что он находится прямо перед вами. Но если увеличить громкость правого динамика, вам покажется, что звук исходит справа. Эта регулировка называется *настройкой баланса* и может помочь вам создать грубо локализованные трехмерные звуки.

Функция задания стереобаланса называется SetPan().

```
HRESULT SetPan(LONG lPan); // значение баланса: от -10000 до 10000
```

Значение баланса также является логарифмическим: величина 0 соответствует центральному расположению, величина -10000 означает, что правый канал ослаблен на -100 dB, а величина 10000 означает, что на -100 dB ослаблен левый канал. Вот пример ослабления звука правого канала на -5 dB:

```
if (FAILED(lpdsbuffer->SetPan(-500)))
  { /* Ошибка */ }
```

Общение с DirectSound

Возможно, вас заинтересует вопрос, существует ли какой-либо способ запросить у DirectSound информацию об аудиосистеме или воспроизводимом звуке, например выяснить, завершилось ли его воспроизведение. Конечно же, существует. В DirectSound имеется ряд функций для выполнения операций подобного рода. Во-первых, существует функция DirectSound для определения возможностей вашего оборудования:

```
HRESULT GetCaps(LPDSCAPS lpDSCaps); // указатель на структуру DSCAPS
```

Эта функция принимает указатель на структуру DSCAPS и заполняет ее. Приведем определение структуры DSCAPS, к которой вы можете обращаться (более полное описание полей можно найти в DirectX SDK, однако и так бо́льшая их часть носит смысловые имена, которые говорят об их назначении).

```
typedef struct _DSCAPS {
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMinSecondarySampleRate;
  DWORD dwMaxSecondarySampleRate;
  DWORD dwPrimaryBuffers;
  DWORD dwMaxHwMixingAllBuffers;
  DWORD dwMaxHwMixingStaticBuffers;
  DWORD dwMaxHwMixingStreamingBuffers;
  DWORD dwFreeHwMixingAllBuffers;
  DWORD dwFreeHwMixingStaticBuffers;
  DWORD dwFreeHwMixingStreamingBuffers;
  DWORD dwMaxHw3DAllBuffers;
  DWORD dwMaxHw3DStaticBuffers;
  DWORD dwMaxHw3DStreamingBuffers;
  DWORD dwFreeHw3DAllBuffers;
```

```
  DWORD dwFreeHw3DStaticBuffers;
  DWORD dwFreeHw3DStreamingBuffers;
  DWORD dwTotalHwMemBytes;
  DWORD dwFreeHwMemBytes;
  DWORD dwMaxContigFreeHwMemBytes;
  DWORD dwUnlockTransferRateHwBuffers;
  DWORD dwPlayCpuOverheadSwBuffers;
  DWORD dwReserved1;
  DWORD dwReserved2;
} DSCAPS, *LPDSCAPS;
```

Вы можете вызвать эту функцию, например, следующим образом:

```
DSCAPS dscaps; // Хранит полученную информацию
if (FAILED(lpds->GetCaps(&dscaps)))
  { /* Ошибка */ }
```

После сбора информации вы можете обратиться к любому из этих полей и определить характеристики своего звукового аппаратного обеспечения. Аналогичная функция имеется и у буферов DirectSound, с тем отличием, что она возвращает структуру DSBCAPS, описывающую буфер.

```
HRESULT GetCaps(LPDSBCAPS lpDSBCaps); // Указатель на
                                      // структуру DSBCAPS
```

Структура DSBCAPS имеет следующее определение:

```
typedef struct _DSBCAPS {
  DWORD dwSize;                 // Размер структуры
  DWORD dwFlags;                // Флаги буфера
  DWORD dwBufferBytes;          // Размер буфера
  DWORD dwUnlockTransferRate; // Частота оцифровки
  DWORD dwPlayCpuOverhead;    // Процессорное время,
                                // необходимое для микширования
} DSBCAPS, *LPDSBCAPS;
```

Теперь покажем, как можно получить информацию об аудиобуфере lpdsbuffer, который мы ранее использовали.

```
DSBCAPS dsbcaps; // Хранит полученную информацию

// Настроить структуру
dsbcaps.dwSize = sizeof(DSBCAPS);

// Получить информацию о буфере
if (FAILED(lpdsbuffer->GetCaps(&dsbcaps)))
  { /* Ошибка */ }
```

Это все, что нужно сделать в данном случае. Кроме описанной, имеются функции считывания значений громкости, баланса, частоты и прочего из любого аудиобуфера, но этот вопрос остается для самостоятельного изучения.

Последняя функция, с которой я хочу вас познакомить, используется для определения состояния аудиобуфера воспроизведения.

```
HRESULT GetStatus(LPDWORD lpdwStatus);
```

Вызовите эту функцию с помощью указателя на интерфейс интересующего вас аудиобуфера, передавая указатель на DWORD, в котором должно быть сохранено состояние буфера.

```
DWORD status; // Используется для хранения состояния
if (FAILED(lpdsbuffer->GetStatus(&status)))
  { /* Ошибка */ }
```

Состояние может быть представлено тремя значениями.

- DSBSTATUS_BUFFERLOST — с буфером случилась неприятность. Наши дела плохи.
- DSBSTATUS_LOOPING — звук воспроизводится циклически.
- DSBSTATUS_PLAYING — в данный момент осуществляется воспроизведение звука. Если этот бит не установлен, звук не воспроизводится.

Чтение звуков с диска

К сожалению, DirectSound не поддерживает непосредственную загрузку звуковых файлов. Я имею в виду, что нет вообще *никакой* поддержки. Нет загрузчика .VOC, нет загрузчика .WAV, ничего нет! Это ужасно неприятно. Приходится писать такую поддержку самостоятельно. Проблема заключается в том, что звуковые файлы чрезвычайно сложны и потребовалось бы полглавы, чтобы дать надлежащее объяснение их форматов. Поэтому я намерен предоставить вам только описание загрузчика .WAV и пояснить в общих чертах, как он работает.

СОВЕТ

В Microsoft разработали собственный загрузчик .WAV, который вы можете при желании использовать. Единственная проблема состоит в том, что данный API не является стандартным и может измениться. Если вас интересует этот вопрос, поищите соответствующие функции в файлах DDUTIL*.CPP в каталоге SAMPLES DirectX SDK.

Формат .WAV

Это формат звуковых файлов Windows, основанный на формате .IFF, изначально созданном компанией Electronic Arts. Сокращение IFF обозначает *Interchange File Format* (формат взаимного файлового обмена). Он является стандартом, позволяющим кодировать файлы различных типов при помощи общей структуры заголовков и данных, допускающей вложение. Формат .WAV применяет именно этот способ кодирования, но, хотя он безупречен и логичен, читать такие файлы стоит большого труда. Вы должны проанализировать массу информации, содержащейся в заголовках (что требует выполнения немалого количества кода), а затем выделить данные, касающиеся звука.

Анализ настолько сложен, что Microsoft создала набор функций (которые получили название интерфейса ввода-вывода мультимедиа (multimedia I/O Interface — MMIO)), для того чтобы помочь вам загружать .WAV-файлы и файлы других подобных типов. Названия всех функций этой библиотеки начинаются с mmio*. Дело в том, что написание программы для чтения .WAV-файлов нельзя назвать простым — это весьма утомительное занятие, никак не связанное с программированием игр. Поэтому я просто приведу код загрузчика .WAV-файлов, снабдив его комментариями, и дам небольшое пояснение. Если вы захотите узнать об этом больше, найдите хорошую книгу, посвященную форматам звуковых файлов.

Чтение .WAV-файлов

Формат .WAV-файлов основан на *разделах* — идентификаторов, форматов и данных. По сути, вы должны открыть .WAV-файл и прочитать всю необходимую информацию о количестве каналов, числе бит на один канал, скорости воспроизведения и т.п., а также о продолжительности воспроизведения звука. И только затем вы загружаете сам звук.

Чтобы облегчить загрузку и воспроизведение звуков, требуется создать API библиотеки работы со звуком, т.е. набор глобальных переменных и функций-оболочек для всей

“начинки” DirectSound. Давайте начнем со структуры данных, в которой будет храниться виртуальный звук и которая будет использоваться вместо более низкоуровневых структур данных DirectSound.

```
// В этой структуре хранится только один звук
typedef struct pcm_sound_typ
{
  LPDIRECTSOUNDBUFFER
    dsbuffer;  // буфер DirectSound, содержащий звук
  int state;    // Состояние звука
  int rate;     // Скорость воспроизведения
  int size;     // Размер звука
  int id;       // Идентификатор звука
} pcm_sound, *pcm_sound_ptr;
```

В этой структуре содержится буфер DirectSound, связанный со звуком, а также копия важной информации о звуке. Теперь создадим массив, предназначенный для сохранения всех звуков, находящихся в системе.

```
pcm_sound sound_fx[MAX_SOUNDS];
  // массив дополнительных аудиобуферов
```

Итак, идея заключается в том, чтобы при загрузке звука выделить для него место в памяти и настроить структуру pcm_sound, чем и занимается функция DSound_Load_WAV().

```
int DSoundLoad_WAV(char*filename,
            int control_flags = DSBCAPS_CTRLDERAULT)
{
// эта функция загружает .WAV-файл, настраивает буфер
// DirectSound и загружает данные в память. Функция
// возвращает идентификатор звука

HMMIO       hwav;          // Дескриптор WAV-файла
MMCKINFO    parent,        // Родительский раздел
            child;         // Дочерний раздел
WAVEFORMATEX  wfmtx;       // Формат
int    sound_id = -1,      // Идентификатор загружаемого звука
       index;              // Переменная цикла
UCHAR  *snd_buffer,        // Временный аудиобуфер
  *audio_ptr_1 = NULL,     // Указатель первого буфера записи
  *audio_ptr_2 = NULL,     // Указатель второго буфера записи
DWORD
  audio_length_1 = 0,      // Длина первого буфера записи
  audio_length_2 = 0,      // Длина второго буфера записи

// шаг первый: имеются ли свободные идентификаторы?
for(index=0; index < MAX_SOUNDS; index++)
{
  // убеждаемся, что этот звук не используется
  if (sound_fx[index].state == SOUND_NULL)
  {
    sound_id = index;
    break;
  } // if
} // for
```

```
// Найден свободный идентификатор?
if (sound_id == -1)
   return (-1);

// Настройка структуры для раздела
parent.ckid             = (FOURCC)0;
parent.cksize           = 0;
parent.fccType          = (FOURCC)0;
parent.dwDataOffset  = 0;
parent.dwFlags      = 0;

// Копирование данных
child = parent

// Открытие WAV-файла
if ((hwav = mmioOpen(filename, NULL,
                MMIO_READ|MMIO_ALLOCBUF))==NULL)
   return (-1);

// Разбор RIFF
parent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
if (mmioDescent(hwav, &parent, NULL, MMIO_FINDRIFF))
{
   // Закрыть файл
   mmioClose(hwav, 0);
   // Возвратить ошибку — раздел не найден
   return (-1);
} // if

// Разбор WAVEfmt
child.ckid = mmioFOURCC('f', 'm', 't', ' ');
if (mmioDescent(hwav,&child,&parent,0))
{
   // Закрыть файл
   mmioClose(hwav, 0);
   // Возвратить ошибку — раздел не найден
   return (-1);
} // if

// Читаем из файла информацию о формате
if (mmioRead(hwav,(char*)&wfmtx,sizeof(wfmtx))
        != sizeof(wfmtx))
{
   // Закрыть файл
   mmioClose (hwav, 0);
   // Возвратить ошибку — нет данных о формате
   return (-1);
} // if

// Убедимся, что формат данных действительно PCM
if (wfmtx.wFormatTag != WAVE_FORMAT_PCM)
{
```

```
    // Закрыть файл
    mmioClose (hwav, 0);
    // Возвратить ошибку — формат данных неверен
    return (-1);
} // if

// Поднимемся на один уровень вверх с тем, чтобы
// получить доступ к разделу данных
if (mmioAscend(hwav,&child,0))
{
    // закрыть файл
    mmioClose(hwav,0);
    // Возвратить ошибку — подняться невозможно
    return (-1);
} // if

// Спуск к разделу данных
child.ckid = mmioFOURCC('d', 'a', 't', 'a');
if (mmioDescend(hwav,&child,&parent,MMIO_FINDCHUNK))
{
    // Закрыть файл
    mmioClose(hwav, 0);
    // Возвратить ошибку — нет данных
    return (-1);
} // if

// Теперь все, что мы должны сделать,— это считать
// данные и настроить буфер DirectSound

// Выделяем память для загрузки звуковых данных
snd_buffer = (UCHAR*)malloc(child.cksize);

// Читаем данные о волне
mmioRead(hwav,(char *)snd_buffer,child.cksize);

// Закрыть файл
mmioClose(hwav,0);

// Задать скорость и размер в структуре данных
sound_fx[sound_id].rate  = wfmtx.nSamplesPerSec;
sound_fx[sound_id].size  = child.cksize;
sound_fx[sound_id].state = SOUND_LOADED;

// Настройка структуры данных формата
memeset(&pcmwf,0,sizeof(WAVEFORMATEX));
pcmwf.wFormatTag     = WAVE_FORMAT_PCM;
pcmwf.wChannels      = 1;             // Моно
pcmwf.nSamplesPerSec  = 11025;   // Всегда эта скорость
pcmwf.nBlockAlign    = 1;
pcmwf.nAvgBytesPerSec =
      pcmwf.NSamplesPerSec*pcmwf.NBlockAlign;
pcmwf.wBitsPerSample  = 8;
pcmwf.cbSize         = 0;
```

```
// Подготовиться к созданию буфера звуков
dsbd.dwSize  = sizeof(DSBUFFERDESC);
dsbd.dwFlags = control_flags|DSBCAPS_STATIC|
          DSBCAPS_LOCSOFTWARE;
dsbd.dwBufferBytes = child.cksize;
dsbd.lpwfxFormat   = &pcmwf;

// Создать аудиобуфер
if (lpds->CreateSoundBuffer(&dsbd,
   &sound_fx[sound_id].dsbuffer,NULL != DS_OK)
{
   // Освободить память
   free (snd_buffer);
   // Возвратить ошибку
   return (-1);
} // if

// Скопировать данные в аудиобуфер
if (sound_fx[sound_id].dsbuffer->Lock(0,
   child.cksize,    (void**)&audio_ptr_1,
   &audio_length_1, (void **) &audio_ptr_2,
   &audio_length_2, DSBLOCK_FROMWRITECURSOR) != DS_OK)
   return (0);

// Копируем первый сегмент кольцевого буфера
memcpy(audio_ptr_1, snd_buffer, audio_length_1);

// Копируем второй сегмент кольцевого буфера
memcpy(audio_ptr_2, (snd_buffer+audio_length_1),
     audio_length_1);

// Разблокируем буфер
if (sound_fx[sound_id].dsbuffer->UnLock(
      audio_ptr_1, audio_length_1,
      audio_ptr_2, audio_length_2) != DS_OK)
   return (0);

// Освобождаем временный буфер
free (snd_buffer);

// Возвращаем идентификатор
return (sound_id)
} // Dsound_Load_WAV
```

Вы просто передаете функции имя файла и стандартные управляющие флаги DirectSound, например DSBCAPS_CTRLDEFAULT или любой другой. Затем происходит следующее.

1. Функция открывает на диске .WAV-файл и извлекает из него информацию о его содержимом.
2. Функция создает буфер DirectSound и заполняет его.
3. Функция сохраняет информацию в свободном элементе массива sound_fx[] и возвращает индекс, который я использую в качестве идентификатора звука.

4. И наконец, остальная часть вашего API будет использовать идентификатор для обращения к звуку, и вы можете делать с ним все, что захотите, например воспроизводить его.

```
// Загрузить звук
int id = Dsound_Load_WAV("test.wav");

// Воспроизведение звука
sound_fxl.lpdsbuffer->Play(0,0,DSBPLAY_LOOPING);
```

Рассмотрите демонстрационную программу DEMO10_3.CPP на прилагаемом компакт-диске (при ее компиляции не забудьте включить в проект DSOUND.LIB и WINMM.LIB). Это завершенная демонстрационная программа DirectSound и функции Dsound_Load_WAV(). Кроме того, она позволяет управлять воспроизведением звука в реальном времени при помощи ползунков. Так что данное приложение позволяет не только понять принципы управления звуком, но и учит использовать ползунки в ваших программах.

Конечно, я намерен использовать все описанные возможности в библиотеке T3DLIB3.CPP, но сначала рассмотрим систему DirectMusic.

Большой эксперимент DirectMusic

DirectMusic — один из самых захватывающих компонентов DirectX, который стал доступным начиная с версии 6.0. Как уже отмечалось, писать программы с использованием цифрового звука чрезвычайно сложно, но работа над воспроизведением MIDI-файлов ничуть не проще. DirectMusic не только воспроизводит MIDI-файлы, но делает и многое другое.

- Поддерживает инструменты DLS (загружаемых звуков). Это означает, что, если вы воспроизводите MIDI-файл при помощи DirectMusic, он будет звучать всегда одинаково, на любом оборудовании.
- Поддерживает сочинение музыки “на ходу”, используя для этого Music Engine, а также позволяет задавать шаблоны, индивидуальные особенности и вариации тональности песен. Затем DirectMusic берет данные вашей композиции и создает музыку в реальном времени.
- Поддерживает число MIDI каналов, ограничиваемое только мощностью вашего компьютера. Обычно MIDI поддерживает одновременно 16 каналов, или 16 отдельных звуков. Имеется 65 535 групп каналов, работающих под управлением DirectMusic, поэтому вы располагаете практически неограниченным числом дорожек, которые могут быть воспроизведены в одно и то же время.
- По возможности использует аппаратное ускорение. По умолчанию установлен программный синтезатор Microsoft, так что звуки имеют такое же высокое качество, как и звуки, полученные в результате табличного или волноводного синтеза.

Единственной плохой новостью о DirectMusic является то, что он так же сложен, как и Direct3D. Я прочитал его документацию (примерно 500 страниц) и могу сказать вам одну вещь: простота не входила в намерения разработчиков. К счастью, все, что нам понадобится от DirectMusic — это воспроизвести MIDI-файл, так что я собираюсь всего лишь продемонстрировать вам, как это можно сделать, а также создать API вокруг DirectMusic, с помощью которого вы сможете загружать и воспроизводить MIDI-файлы. Кроме того, как уже упоминалось при рассмотрении DirectX 8.0, системы DirectSound и DirectMusic были объединены (а не просто включены) в одну — DirectX Audio. В результате появился новый интерфейс IDirectSound8 и новый интерфейс системы DirectMusic, но эти интер-

фейсы никак не способствуют достижению наших целей, а лишь усложняют многие вещи. Поэтому просто не будем их рассматривать.

Архитектура DirectMusic

DirectMusic является достаточно большой системой, поэтому я не собираюсь подробно вникать во все ее детали: это тема для отдельной книги. Однако я должен хотя бы упомянуть интерфейсы, с которыми вам придется работать. Они показаны на рис. 10.12.

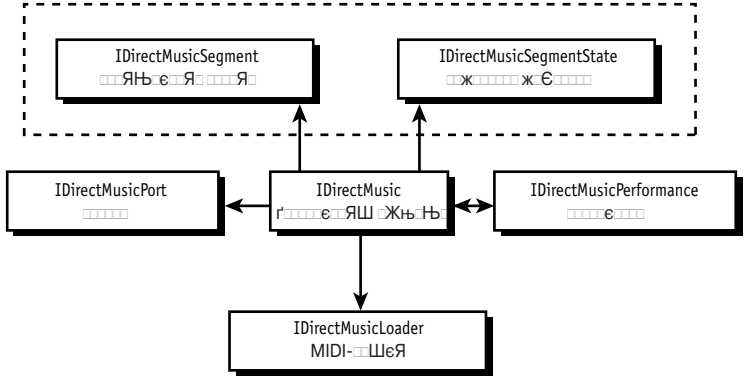


Рис. 10.12. Основные интерфейсы DirectMusic

Приведем краткое описание этих интерфейсов.

IDirectMusic. Это основной интерфейс DirectMusic, но, в отличие от DirectDraw и DirectSound, он не обязателен при использовании DirectMusic. Он создается по умолчанию и оказывается скрытым, когда вы создаете исполняющий объект DirectMusic (и слава Богу!).

IDirectMusicPerformance. Это основной интерфейс, который непосредственно касается вас. Исполняющий объект управляет воспроизведением всех музыкальных данных. Кроме того, при создании этот интерфейс создает объект IDirectMusic.

IDirectMusicLoader. Этот интерфейс используется для загрузки всех данных, включая MIDI, DLS и т.д. Он используется для загрузки MIDI-файлов с диска. Так что у вас есть загрузчик MIDI — а это большая помощь!

IDirectMusicSegment. Этот интерфейс представляет раздел музыкальных данных; каждый загруженный MIDI-файл будет представлен с помощью этого интерфейса.

IDirectMusicSegmentState. Этот интерфейс связан с сегментом, но он имеет отношение не к данным, а к текущему состоянию сегмента.

IDirectMusicPort. Это именно то место, куда направляется поток цифровых данных, представляющих вашу MIDI-музыку. В большинстве случаев это будет Microsoft Software Synthesizer, но вы всегда можете перечислить другие возможные порты с аппаратным ускорением.

Вообще говоря, DirectMusic — это преобразователь реального времени, который преобразует формат MIDI в цифровой и обладает возможностями обработки цифрового сигнала. Как уже упоминалось при рассмотрении вопроса о MIDI в рамках системы DirectSound, проблема, связанная с MIDI, заключается в том, что звуковой файл, будучи зависимым от аппаратного обеспечения и инструментальных средств, может по-разному воспроизводиться на разных машинах. DirectMusic решает эту проблему, используя чисто цифровые образцы инструментов в виде DLS-файлов. Поэтому всякий раз, когда вы создаете композицию, вы либо можете использовать задаваемый по умолчанию DLS-файл, либо создать собственный файл инструмента. Вся хитрость в том, что инструменты являются цифровыми по

своей природе и поставляются вместе с вашей музыкой. Цифровые звуки всегда воспроизводятся через цифроаналоговый преобразователь, поэтому такая музыка всегда звучит одинаково. Чтобы понять, как это работает, обратимся к рис. 10.13.

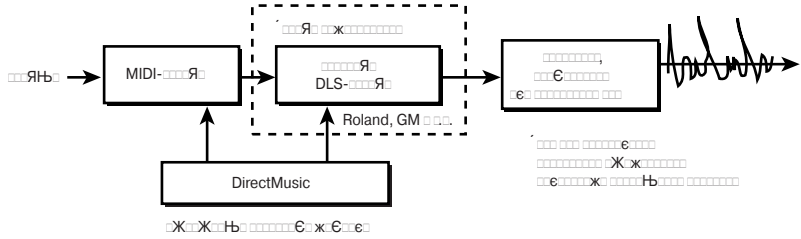


Рис. 10.13. DirectMusic опирается на цифровые образцы

НА ЗАМЕТКУ

Задаваемые по умолчанию DLS-инструменты, имеющиеся на каждой машине и загружаемые при помощи DirectMusic, — это Roland GM/GS (General MIDI). Они звучат великолепно, но вы никоим образом не можете изменить их: Roland не хочет, чтобы вы их ухудшили!

Конечно, все это выглядит сложнее, чем хотелось бы. Но сложность хороша потому, что позволяет внедрять новые технологии и всевозможные новшества, которые составляют основу DirectMusic.

Запуск DirectMusic

DirectMusic — это первый компонент системы DirectX, который является абсолютно чистым представителем COM. Это значит, что здесь нет никаких вспомогательных функций из библиотеки импорта, позволяющих создавать для вас объекты COM. Увы! Здесь вы должны создавать COM-объекты сами посредством вызова COM-библиотеки. Поэтому любое приложение нуждается только в файлах заголовков DirectMusic и нет никаких библиотечных файлов импорта. В состав DirectMusic входят следующие заголовочные файлы:

```
dmksctrl.h
dmusici.h
dmusicc.h
dmusicf.h
```

Просто удостоверьтесь в том, что они включены в исходные файлы вашего приложения, а COM позаботится обо всем остальном. Давайте рассмотрим всю последовательность действий.

Инициализация COM

Сначала вы должны инициализировать COM посредством вызова функции `CoInitialize()`.

```
// Инициализация COM
if (FAILED(CoInitialize(NULL)))
{
  // Завершение приложения
  return (0);
} // if
```

Эти действия следует выполнить в начале вашего приложения, до каких бы то ни было прямых вызовов COM.

Создание исполняющей системы

Следующий шаг — создание основного интерфейса, который представляет собой исполняющую систему DirectMusic. Создание этого интерфейса приводит к одновременному созданию внутреннего интерфейса IDirectMusic, но нам он не нужен и поэтому остается скрытым. Для создания интерфейса на базе COM используйте функцию CoCreateInstance(), применяя идентификатор интерфейса и идентификатор класса; не забудьте после этого сохранить указатель интерфейса.

```
// Диспетчер исполняющей системы DirectMusic
IDirectMusicPerformance *dm_perf = NULL;

// Создание исполняющей системы
if (FAILED(CoCreateInstance(CLSID_DirectMusicPerformance,
                  NULL,
                  CLSCTX_INPROC,
                  IID_IDirectMusicPerformance,
                  (void**)&dm_perf)))
{
  // Возврат нулевого значения
  return (0);
} // if
```

Этот код выглядит немного странно, но в пределах разумного. После этого вызова указатель dm_perf готов к работе, и вы можете обращаться к функциям данного интерфейса. Первый вызов, который вы должны сделать, — инициализация исполняющей системы при помощи вызова IDirectMusicPerformance::Init().

```
HRESULT Init(IDirectMusic** ppDirectMusic,
       LPDIRECTSOUND pDirectSound,
       HWND      hWnd);
```

Параметр ppDirectMusic — это адрес интерфейса IDirectMusic, если вы создаете его явным образом; если нет — установите его значение равным NULL. Параметр pDirectSound — это указатель на объект IDirectSound.

ВНИМАНИЕ

Это очень важно, поэтому прочитайте внимательно. Если вы хотите использовать DirectSound и DirectMusic одновременно, в таком случае сначала запустите DirectSound и затем передайте объект IDirectSound при обращении к функции Init(). Если же вы используете только DirectMusic, передайте NULL, и DirectMusic создаст объект IDirectSound. Необходимость этого объясняется тем, что DirectMusic, в конечном счете, работает посредством DirectSound; об этом свидетельствует и схема на рис. 10.14.

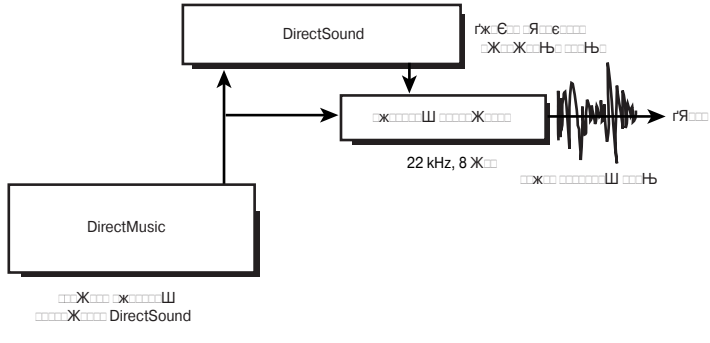


Рис. 10.14. Взаимоотношения DirectMusic и DirectSound

Мораль сей басни такова: вы должны передать либо указатель на основной объект DirectSound, либо NULL, если DirectSound явным образом не используется. И наконец, вы должны передать функции дескриптор окна. Это очень просто — судите сами:

```
// Инициализировать исполняющую систему,
// при использовании DirectSound передать соответствующий
// объект; если нет, следует передать NULL
if (FAILED(dm_perf->Init(NULL,NULL,main_window_handle)))
{
  return(0);
  //Ошибка — исполняющая система не инициализирована
} // if
```

Добавление порта

Следующий шаг подготовки и запуска DirectMusic заключается в создании порта, куда будет направлен поток цифровых данных. Если хотите, можно попросить DirectMusic перечислить все корректные порты в системе, а можно просто использовать Microsoft Software Synthesizer, задаваемый по умолчанию (в этом мой стиль — решать задачу как можно проще). Чтобы добавить порт к исполняющей системе, воспользуйтесь вызовом функции

```
HRESULT AddPort(IDirectMisicPort *pPort);
```

Здесь pPort — это указатель на созданный ранее порт, который вы хотите использовать для воспроизведения. Если передать в качестве этого параметра NULL, то будет использоваться программный синтезатор по умолчанию.

```
// Добавить порт к исполняющей системе
if (FAILED(dm_perf->AddPort(NULL)))
{
  return (0); // Ошибка – порт не инициализирован
} // if
```

Загрузка MIDI

Следующий шаг настройки DirectMusic — это создание объекта IDirectMusicLoader, позволяющего загружать MIDI-файлы. Эта операция вновь выполняется посредством низкоуровневого вызова COM, но это не так уж плохо.

Создание загрузчика

Загрузчик создается при помощи следующего кода:

```
// Загрузчик DirectMusic
IDirectMusicLoader*dm_loader = NULL;

// Создание загрузчика объектов
if (FAILED(CoCreateInstance(
            CLSID_DirectMusicLoader,
            NULL,
            CLSCTX_INPROC,
            IID_IDirectMusicLoader,
            (void**)&dm_loader)))
{
  // ошибка
  return (0);
} // if
```

Довольно интересно то, что при этом были созданы некоторые внутренние интерфейсы (включая объекты IDirectMusic и IDirectMusicPort), но вы об этом даже не узнали. В большинстве случаев вам вообще не нужно обращаться к функциям этих интерфейсов.

Загрузка MIDI-файла

Чтобы загрузить MIDI-файл, необходимо сообщить загрузчику, что*го* именно (тип файла) и где нужно найти, а затем дать ему команду создать сегмент и загрузить в него файл. Я создал функцию, которая делает это при помощи некоторой структуры данных, что вам сейчас и покажу. Вот структура данных, предназначенная для хранения музыкальных MIDI-сегментов (в DirectMusic принято называть разделы данных *сегментами*):

```
typedef struct DMUSIC_MIDI_TYP
{
  IDirectMusicSegment     *dm_segment;
                  // сегмент DurectMusic
  IDirectMusicSegmentState *dm_segment;
                  // состояние сегмента
  int               id;
                  // идентификатор сегмента
  int               state;
                  // состояние MIDI-композиции
}  DMUSIC_MIDI,  *DMUSIC_MIDI_PTR;
```

Она используется для хранения каждого MIDI-сегмента. Так как в течение всей игры вы можете использовать множество разных MIDI-произведений, создадим для них массив.

```
#define DM_NUM_SEGMENTS 64 // число MIDI-сегментов, которые
                                  // могут храниться в памяти
DMUSIC_MIDI dm_midi[DM_NUM_SEGMENTS];
```

Теперь рассмотрим функцию DMusic_Load_MIDI(). Она снабжена комментариями, на которые я рекомендую обратить внимание. Кроме того, обратите внимание, что в функции используются *широкие* символы (WCHAR, или, что то же самое, стандартный тип C++ wchar_t).

```
int DMusic_Load_MIDI(char *filename)
{
  // Эта функция загружает MIDI-сегмент
  DMUS_OBJECTDESC    Objdesc;
  HRESULT          hr;
  IDirectMusicSegment* pSegment = NULL;
  int              index;  // переменная цикла

  // Поиск свободной области для MIDI-сегмента
  int  id = -1;
  for(index = 0; index<DM_NUM_SEGMENTS; index++)
  {
    // Эта не занята
    if (dm_midi[index].state == MIDI_NULL)
    {
      // Получаем идентификатор
      id = index;
      break;
    } // if
  } // for
```

```
// Найден идентификатор?
if (id ==-1) return(-1);

// Находим текущий рабочий каталог
char  szDir [_MAX_PATH];
WCHAR wszDir[_MAX_PATH];
if (_getcwd(szDir,_MAX_PATH)==NULL)
{
   return (-1);
} // if
MULTI_TO_WIDE(wszDir,szDir);

// Сообщаем загрузчику, где искать файлы
hr = dm_loader->SetSearchDirectory(
      GUID_DirectMusicAllTypes,wszDir,FALSE);
if (FAILED(hr))
{
   return(-1);
}  // if

// Преобразуем имя файла
WCHAR wfilename[_MAX_PATH];
MULTI_TO_WIDE(wfilename,filename);

// Настраиваем описание объекта
DO_INIT_STRUCT(ObjDesc);
ObjDesc.guidClass = CLSID_DirectMusicSegment;
wcscpy(ObjDesc.wszFileName,wfilename);
ObjDesc.dwValidData = DMUS_OBJ_CLASS|DMUS_OBJ_FILENAME;

// Загружаем объект и запрашиваем его интерфейс
// IDirectMusicSegment. Это делается путем вызова
// IDirectMusicLoader::GetObject. Заметим, что при
// загрузке объекта инициализируются дорожки и делается
// все необходимое для подготовки MIDI-данных
// к воспроизведению
hr = dm_loader->GetObgect(&ObjDesc,
      IID_IDirectMusicSegment,(void**)&pSegment);
if (FAILED(hr)) return(-1);

// Убедимся, что сегмент воспроизводится как стандартный
// MIDI-файл. Теперь нужно задать параметр для группы
// дорожек при помощи IDirectMusicSegment::SetParam
hr =p_Segment->SetParam(GUID_StandardMIDIFile, -1,
                  0,0,(void*)dm_perf);
if (FAILED(hr)) return (-1);

// Этот шаг был необходим, поскольку DirectMusic
// по-разному обрабатывает стандартные MIDI-файлы
// и MIDI-данные, созданные специально для DirectMusic.
// Параметр GUID_StandardMIDIFile должен быть
// определен до того, как будут загружены инструменты.


// На следующем этапе загружаются инструменты. Эта
// операция необходима для воспроизведения даже простого
```

```
  // MIDI-файла, поскольку задаваемый по умолчанию
  // программный синтезатор требует DLS-данных.
  // Если пропустить этот шаг, MIDI-файл будет
  // воспроизводиться "молча". Мы используем вызов
  // SetParam для сегмента, передавая параметр
  // GUID_DownLoad
  hr =pSegment->SetParam(GUID_DownLoad, -1,
              0,0,(void*)dm_perf);
  if (FAILED(hr)) return(-1);

  // Теперь у нас есть загруженный MIDI и корректный объект
  dm_midil.dm_segment  = pSegment;
  dm_midil.dm_segstate = NULL;
  dm_midil.state       = MIDI_LOADED;

  // Возврат идентификатора
  return (id);
} // DMusic_Load_MIDI
```

Эта функция ищет свободные области в вашем массиве dm_midi[] для загрузки нового MIDI-сегмента, задает путь поиска, создает сегмент, загружает его и завершается. Она принимает в качестве параметра имя MIDI-файла и возвращает индекс элемента массива, в котором содержится сегмент.

Работа с MIDI-сегментами

Интерфейс IDirectMusicSegment, который представляет загруженный MIDI-сегмент, имеет ряд методов. Если вас всерьез интересует этот вопрос, то подробности вы можете найти в SDK. Однако нам не обойтись без двух функций — воспроизведения и остановки. Но они являются частью интерфейса IDirectMisicPerformance, а не IDirectMisicSegment. Если подумать, это не лишено смысла: исполняющий объект похож на диспетчера, от взгляда которого ничто не должно ускользнуть.

Воспроизведение MIDI-сегмента

Предположим, вы загрузили сегмент при помощи вызова DMusic_Load_MIDI() или сделали это вручную и dm_segment — указатель на интерфейс этого сегмента. Тогда для его воспроизведения при помощи исполняющего объекта можно использовать функцию IDirectMusicPerformance::PlaySegment().

```
HRESULT PlaySegment(
 IDirectMusicSegment*
       pSegment,          // Воспроизводимый сегмент
 DWORD    dwFlags,        // Управляющие флаги
 _int64   i64StartTime,   // Время начала воспроизведения
 IDirectMusicSegmentState**
       ppSegmentState);// Сохраненное состояние
```

В общем случае укажите 0 для управляющих флагов и времени начала воспроизведения. Внимания требуют только параметры сегмента и его состояния. Вот пример воспроизведения dm_segment и сохранения состояния в переменной dm_segstate:

```
dm_perf->PlaySegment(dm_segment, 0, 0, &dm_segstate);
```

Здесь dm_segstate представляет собой переменную типа IDirectMusicSegmentState и используется для отслеживания воспроизведения сегмента.

Остановка воспроизведения MIDI-сегмента

Чтобы остановить воспроизведение сегмента, необходимо использовать функцию IDirectMusicPerformance::Stop().

```
HRESULT Stop(
  IDirectMusicSegment* pSegment, // Останавливаемый сегмент
  IDirectMusicSegmentState*
              pSegmentState,      // Состояние сегмента
  MUSIC_TIME       mtTime,        // Время останова
  DWORD          wdFlags);        // Управляющие флаги
```

Как и в случае с функцией Play(), вам не нужно заботиться о большинстве параметров, уделяя все внимание только самому сегменту. Вот пример останова воспроизведения dm_segment:

```
dm_perf->Stop(dm_segment, NULL, 0, 0 );
```

Если вы хотите остановить *все* воспроизводящиеся сегменты, задайте в качестве параметра dm_segment значение NULL.

Проверка состояния MIDI-сегмента

Если вам хочется узнать, не завершилось ли воспроизведение сегмента, воспользуйтесь функцией IDirectMusicPerformance::IsPlaying(). Она принимает в качестве параметра интересующий вас сегмент и возвращает значение S_OK, если сегмент все еще воспроизводится.

```
if (dm_perf->IsPlaying(dm_segment,NULL) == S_OK)
  { /* Еще воспроизводится */ }
else
  { /* Не воспроизводится */ }
```

Освобождение MIDI-сегмента

По завершении работы с сегментом вы должны освободить ресурсы. На первом этапе посредством вызова функции IDirectMusicSegment::SetParam() выгружаются DLS-данные, а затем при помощи функции Release() освобождается непосредственно указатель на интерфейс.

```
// Выгрузить данные инструмента
dm_segment->SetParam(GUID_Unload,-1,0,0,(void*)dm_perf);

// Освободить сегмент
dm_segment->Release();
dm_segment->NULL; // Для безопасности
```

Завершение работы с DirectMusic

По завершении работы с DirectMusic необходимо закрыть и освободить исполняющий объект, загрузчик и все сегменты. И наконец, нужно закрыть COM, если это не было сделано где-нибудь в другом месте. Данный процесс выглядит примерно так:

```
// Если воспроизводится какая-либо музыка, остановите ее
// (это не обязательно, поскольку музыка все равно
// остановится, когда будут выгружены инструменты или
// будет закрыта исполняющая система)
if (dm_perf)
  dm_perf->Stop(NULL,NULL,0,0);
```

```
// Удалить все MIDI, которые еще не были удалены


// Закрыть и освободить исполняющий объект
if (dm_perf)
  {
    dm_perf->CloseDown();
    dm_perf->Release();
  } // if

// Освободить объект загрузчика
 if (dm_loader)
   dm_loader->Release();

// Освободить COM
CoUninitialize();
```

Пример использования DirectMusic

В качестве примера использования системы DirectMusic (без DirectSound или любого другого компонента DirectX) на прилагаемом компакт-диске находится демонстрационная программа DEMO10_4.CPP. Она загружает единственный MIDI-файл и воспроизводит его. Рассмотрите этот пример и поэкспериментируйте с его исходным кодом. Посмотрите, насколько просто решается поставленная задача при помощи библиотеки T3DLIB3.CPP.

Звуковая библиотека T3DLIB3

Все описанные выше технологии собраны в небольшой библиотеке T3DLIB3, которую вы можете использовать при создании игр. Эта библиотека состоит из двух основных файлов:

- T3DLIB3.CPP — основной исходный файл на языке C/C++;
- T3DLIB3.H — заголовочный файл.

Чтобы данная библиотека могла работать в составе вашего приложения, в проект необходимо включить библиотеку импорта DirectSound — DSOUND.LIB. Поскольку DirectMusic представляет собой чистый COM, этот компонент не имеет своей библиотеки импорта, поэтому никакой библиотеки типа DMUSIC.LIB включать в проект не требуется. Тем не менее в исходный текст проекта должны включаться следующие заголовочные файлы:

```
DSOUND.H
DMKSCTRL.H
DMUSICI.H
DMUSICC.H
DMUSICF.H
```

Рассмотрим основные элементы заголовочного файла T3DLIB3.H.

Заголовочный файл

Заголовочный файл T3DLIB.H содержит типы, макросы и внешние переменные библиотеки T3DLIB3.CPP.

Вот макроопределения из упомянутого заголовочного файла:

```
// Количество MIDI-сегментов, хранящихся в памяти
#define DM_NUM_SEGMENTS 64

// Определения состояний MIDI-объекта
#define MIDI_NULL   0    // MIDI-объект не загружен
#define MIDI_LOADED 1  // MIDI-объект загружен
#define MIDI_PLAYING 2 // MIDI-объект загружен и воспроизводится
#define MIDI_STOPPED 3 // MIDI-объект загружен, но не воспроизводится

#define MAX_SOUNDS  256 // максимальное количество звуков в системе

// Определения состояний цифрового аудиообъекта
#define SOUND_NULL   0 // " "
#define SOUND_LOADED 1
#define SOUND_PLAYING 2
#define SOUND_STOPPED 3
```

Макросов в файле немного:

```
// Преобразование диапазона 0 — 100 в шкалу децибел
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100—volume)))

// Преобразование из многобайтового формата в Unicode
#define MULTI_TO_WIDE(x,y) MultiByteToWideChar(CP_ACP,\
            MB_PRECOMPOSED,y,-1,x,_MAX_PATH)
```

Далее следуют определения типов.

Типы

Первый тип представляет объект DirectSound. Звуковые подсистемы бывают только двух типов: один — для хранения цифрового образца, второй — для хранения MIDI-сегмента.

```
// Хранение цифрового звука
typedef struct pcm_sound_typ
{
  LPDIRECTSOINDBUFFER
    dsbuffer; // Буфер DirectSound, содержащий звук
  int state;   // Состояние звука
  int rate;    // Скорость воспроизведения
  int size;    // Размеры звука
  int id;      // Идентификатор звука
} pcm_sound, *pcm_sound_ptr;

// MIDI-сегмент DirectMusic
typedef struct DMUSIC_MIDI_TYP
{
  IDirectMusicSegment
       *dm_segment;  // Сегмент DirectMusic
  IDirectMusicSegmentState
       *dm_segment;  // Состояние сегмента
  int    id;             // Идентификатор сегмента
  int    state;          // Состояние MIDI-композиции
} DMUSIC_MIDI, *DMUSIC_MIDI_PTR;
```

Глобальные переменные

Библиотека T3DLIB3 содержит ряд глобальных переменных. Начнем с глобальных переменных DirectSound.

```
LPDIRECASOUND lpds;      // Указатель интерфейса DirectSound
DSBUFFERDESC  dsbd;      // Описание DirectSound
DSCAPS      dscaps;      // Информация о DirectSound
HRESULT     dsresult;    // Общий результат DirectSound
DSBCAPS     dsbcaps;     // Информация о буфере DirectSound
pcm_sound    sound_fx[MAX_SOUNDS];
                          // Массив звуковых буферов
WAVEFORMATEX pcmwf;  // Обобщенная структура формата волны
```

Теперь рассмотрим глобальные переменные DirectMusic.

```
// Глобальные переменные DirectMusic
// Диспетчер исполняющей системы DirectMusic
IDirectMusicPerformance *dm_perf;
IDirectMusicLoader     *dm_loader;  // Загрузчик DirectMusic

// Массив MIDI-объектов DirectMusic
DMUSIC_MIDI         dm_midi[DM_NUM_SEGMENTS];
int               dm_active_id; // активный сегмент
```

Вы не должны непосредственно работать со всеми этими глобальными переменными, за исключением необходимости прямого доступа к интерфейсам. Все управление берет на себя API, а глобальные переменные существуют только на тот случай, если вы захотите "вырвать" их из общего контекста.

Библиотека состоит из двух частей — для работы с DirectSound и DirectMusic. Вначале рассмотрим работу с DirectSound, а затем — с DirectMusic.

DirectSound API

DirectSound может быть простой или сложной, в зависимости от того, как именно вы ее используете. Если вы желаете получить API, в который будут "завернуты" все возможности, то в конечном итоге придете к использованию большинства функций DirectSound в их исходном виде. Но если вам нужно только инициализировать DirectSound и загружать и воспроизводить звуки специального формата, то можно обойтись всего лишь несколькими функциями. Поэтому я постарался ограничить библиотеку, внеся в нее только самое необходимое. Кроме того, в ней вы можете обращаться к звуку при помощи идентификатора (то же сделано и для DirectMusic), который вы получаете при загрузке этого звука. Разработанный API поддерживает следующие функции:

- инициализацию и закрытие DirectSound одним вызовом;
- загрузку .WAV-файлов в формате 11 kHz, 8-бит, моно;
- воспроизведение загруженного звукового файла;
- останов звука;
- проверку состояния воспроизведения звука;
- изменение громкости, скорости воспроизведения или стереобаланса звука;
- удаление звуков из памяти.

Рассмотрим каждую из этих функций в отдельности.

НА ЗАМЕТКУ Если не указано иное, все функции возвращают TRUE (1) в случае успешного завершения и FALSE (0) в противном случае.

Прототип функции:

```
int DSound_Init(void);
```

Назначение:

Функция DSound_Init() инициализирует DirectSound. Она создает COM-объект DirectSound, задает уровень приоритета и т.д. Если вы хотите использовать звук, просто вызовите эту функцию в начале вашего приложения.

```
if (!DSound_Init(void))
  { /* Ошибка */ }
```

Прототип функции:

```
int DSound_Shutdown(void);
```

Назначение:

Функция DSound_Shutdown() закрывает и освобождает все COM-интерфейсы, созданные во время работы DSound_Init(). Однако DSound_Shutdown не освобождает всю память, выделенную для звуков. Вы должны сделать это самостоятельно при помощи другой функции.

```
if (!DSound_Shutdown(void))
  { /* Ошибка */ }
```

Прототип функции:

```
int DSound_Load_WAV(char *filename);
```

Назначение:

Функция DSound_Load_WAV() создает буфер DirectSound, загружает файл звуковых данных в память и подготавливает звук к воспроизведению. Эта функция принимает полный путь и имя загружаемого звукового файла (включая расширение .WAV) и загружает файл с диска. В случае успешного завершения функция возвращает неотрицательное число — идентификатор звука. Вы должны сохранить этот идентификатор, который используется в качестве дескриптора для обращения к звуку. Если функция не может найти файл или загружено слишком много звуков, она возвратит значение -1. Вот пример загрузки .WAV-файла с именем FIRE.WAV.

```
int fire_id = DSound_Load_WAV("FIRE.WAV");

// Проверка на наличие ошибки
if (fire_id == -1)
  { /* Ошибка */ }
```

Конечно, как вы будете хранить идентификаторы — дело ваше. Вы можете использовать для этого массив или что-нибудь еще.

И наконец, вам может быть интересно, где находятся звуковые данные и как к ним обратиться непосредственно. Если вам действительно необходимо это сделать, вы можете получить доступ к элементам массива sound_fx[], тип которых pcm_sound, используя в качестве индекса дескриптор звука, который вам возвратила любая из функций загрузки. Например, вот как можно было бы осуществить доступ к буферу DirectSound для получения звука с идентификатором sound_id:

```
sound_fx[sound_id].dsbuffer
```

Прототип функции:

```
int DSound_Replicate_Sound(int Source_id);
      // идентификатор копируемого звука
```

Назначение:

Функция DSound_Replicate_Sound() используется для копирования звука без копирования памяти, в которой он хранится. Допустим, у вас есть звук выстрела и вы хотите выстрелить три раза подряд. Единственный способ, который сразу приходит на ум, — загрузить три копии звука выстрела в три разных буфера DirectSound, но это приведет к чрезмерному расходованию памяти.

Есть и другое решение — можно создать копию звукового буфера, не осуществляя при этом реального копирования звуковых данных. Вместо дублирования данных вы просто указываете на них, и DirectSound может использовать одни и те же данные для разных звуков. Если вы хотите воспроизвести выстрел восемь раз, загрузите звук выстрела один раз, сделайте семь его копий и в сумме запросите восемь уникальных идентификаторов. Продублированные звуки работают точно так же, как и нормальные, с тем исключением, что вместо использования функции DSound_Load_WAV() для их загрузки и создания вы копируете их при помощи функции DSound_Replicate_Sound(). Если все еще непонятно, то вот пример создания восьми одинаковых выстрелов:

```
int gunshot_ids[8]; // Здесь хранятся идентификаторы

// Загружаем основной звук
gunshot_ids[0] = DSound_Load_WAV("GUNSHOT.WAV");

// Делаем копии
for(int index = 1; index < 8; index++)
  gunshot_ids[index] =
    DSound_Replicate_Sound(gunshot_ids[0]);

// Используйте gunshot_ids[0..7] как хотите —
// это один и тот же звук
```

Прототип функции:

```
int DSound_Play_Sound(int id,   // Идентификатор звука
      int flags  = 0,           // 0 или DSBPLAY_LOOPING
      int volume = 0,           // Не используется
      int rate   = 0,           // Не используется
      int pan    = 0);          // Не используется
```

Назначение:

Функция DSound_Play_Sound() воспроизводит звук, загруженный раннее. Вы просто указываете идентификатор звука, а также флаги воспроизведения: 0 для однократного воспроизведения звука или DSBPLAY_LOOPING для циклического, и звук начнет воспроизводиться. Если же звук уже воспроизводится, его воспроизведение начнется заново. Вот пример использования этой функции:

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id, 0);
```

Поскольку все параметры функции, кроме первого, имеют значения по умолчанию, значение флагов, равное 0, можно не указывать.

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id);
```

В любом из этих фрагментов звук FIRE.WAV будет воспроизведен однократно. Чтобы задать циклическое воспроизведение, следует в качестве флага передать значение DSBPLAY_LOOPING.

Прототип функции:

```
int DSound_Stop_Sound(int id);
int DSound_Stop_All_Sounds(void);
```

Назначение:

Функция DSound_Stop_Sound() применяется для остановки воспроизведения некоторого звука (если он воспроизводится в данный момент). Вы просто задаете идентификатор звука — и все. Функция DSound_Stop_All_Sounds() остановит звучание всех воспроизводимых в данный момент звуков. Вот пример останова звука fire_id:

```
int DSound_Stop_Sound(fire_id);
```

Желательно, чтобы перед завершением работы программы вы остановили воспроизведение всех звуков. Это можно сделать посредством отдельных вызовов DSound_Stop_Sound() для каждого звука в отдельности или при помощи вызова функции DSound_Stop_All_Sounds().

Прототип функции:

```
int DSound_Delete_Sound(int id); // Идентификатор звука
int DSound_Delete_All_Sounds(void);
```

Назначение:

Функция DSound_Delete_Sound() удаляет звук из памяти и освобождает связанный с ним буфер DirectSound. Если звук воспроизводится, вначале будет остановлено его воспроизведение. Функция DSound_Delete_All_Sounds() удаляет все загруженные ранее звуки. Вот пример удаления звука fire_id:

```
int DSound_Delete_Sound(fire_id);
```

Прототип функции:

```
int DSound_Status_Sound(int id);
```

Назначение:

Функция DSound_Status_Sound() проверяет состояние загруженного звука, используя для обращения его идентификатор. Все, что вам нужно сделать, — передать в функцию идентификатор звука, и она возвратит одно из следующих значений:

- DSBSTATUS_LOOPING — звук в данный момент воспроизводится в цикличном режиме;
- DSBSTATUS_PLAYING — звук в данный момент воспроизводится в однократном режиме.

Если значение, возвращенное функцией DSound_Status_Sound(), не является одной из этих констант, значит, звук не воспроизводится. Ниже приведен фрагмент кода, который ожидает до тех пор, пока не закончится воспроизведение звука, а затем удаляет его.

```
// Инициализируем DirectSound
DSound_DSound_Init();

// Загружаем звук
int fire_id = DSound_Load_WAV("FIRE.WAV")

// Однократно воспроизводим звук
```

```
DSound_Play_Sound (fire_id);

// Ожидаем завершения воспроизведения
while(DSound_Sound_Status(fire_id)&
   (DSBSTATUS_LOOPING|DSBSTATUS_PLAYING));

// Удаляем звук
DSound_Delete_Sound(fire_id);

// Закрываем DirectSound
DSound_DSound_Shutdown();
```

Не правда ли, все намного проще, чем несколько сотен (или около того) строк кода, которые были бы необходимы для того, чтобы сделать все это при помощи DirectSound вручную.

Прототип функции:

```
int DSound_Set_Sound_Volume(int id,  // Идентификатор звука
                int vol); // Громкость от 0 до 100
```

Назначение:

Функция DSound_Set_Sound_Volume() изменяет громкость воспроизведения звука в реальном времени. Передайте ей идентификатор звука, а также значение от 0 до 100, и звук тотчас изменится. Приведем пример уменьшения громкости звука на 50% по сравнению с первоначальной.

```
DSound_Set_Sound_Volume(fire_id,50);
```

Вы всегда можете опять сделать громкость звука равной 100%.

```
DSound_Set_Sound_Volume(fire_id,100);
```

Прототип функции:

```
int DSound_Set_Sound_Freq(int id,  // идентификатор звука
              int freq);           // новая скорость воспроизведения—
                                   // от 0 до 100000
```

Назначение:

Функция DSound_Set_Sound_Freq() изменяет частоту воспроизведения звука. Поскольку при загрузке всех звуков их частота должна быть равна 11 kHz, вот каким образом можно удвоить скорость воспроизведения:

```
DSound_Set_Sound_Freq(fire_id,22050);
```

А так — сделать звук похожим на голос Дарта Вадера:

```
DSound_Set_Sound_Freq(fire_id,6000);
```

Прототип функции:

```
int DSound_Set_Sound_Pan(
  int id,  // Идентификатор звука
  int pan); // Значение стереобаланса от -10000 до 10000
```

Назначение:

Функция DSound_Set_Sound_Pan() задает относительную интенсивность звука на правом и левом динамиках. Значение -10000 — это максимальная громкость звучания левого, а +10000 — правого канала. Если вы хотите задать равную мощность обоих динамиков, передайте функции значение 0. Вот пример кода, когда работает только правый динамик:

```
int DSound_Set_Sound_Pan(fire_id, 10000);
```

DirectMusic API

API DirectMusic еще проще, чем API DirectSound. Я создал функции, предназначенные для инициализации DirectMusic и создания всех COM-объектов, что позволит вам сосредоточиться на загрузке и воспроизведении MIDI-файлов. Список функциональных возможностей библиотеки выглядит так:

- инициализация и закрытие DirectMusic одним вызовом;
- загрузка MIDI-файлов с диска;
- воспроизведение MIDI-файла;
- останов MIDI-файла, воспроизводящегося в данный момент;
- проверка состояния воспроизведения MIDI-сегмента;
- автоматическое подсоединение к DirectSound (если эта подсистема уже инициализирована);
- удаление MIDI-сегментов из памяти.

Рассмотрим каждую функцию отдельно.

НА ЗАМЕТКУ Если не указано иное, все функции возвращают TRUE (1) в случае успешного завершения и FALSE (0) в противном случае.

Прототип функции:

```
int DMusic_Init(void);
```

Назначение:

Функция DMusic_Init() инициализирует систему DirectMusic и создает все необходимые COM-объекты. Вызов этой функции выполняется до любого другого обращения к библиотеке DirectMusic. Кроме того, если вы хотите использовать DirectSound, убедитесь в том, что DirectSound инициализирована до вызова функции DMusic_Init(). Вот пример использования этой функции:

```
if (!DMusic_Init())
  { /* Ошибка */ }
```

Прототип функции:

```
int DMusic_Shutdown(void);
```

Назначение:

Функция DMusic_Shutdown() закрывает всю подсистему DirectMusic. Она освобождает все COM-объекты и выгружает все загруженные MIDI-сегменты. Вызывайте эту функцию в конце приложения, но перед обращением к функции, закрывающей DirectSound (если вы работаете с DirectSound независимо от DirectMusic). Вот пример использования этой функции:

```
if (!DMusic_Shutdown())
  { /* Ошибка */ }
// После этого закрываем DirectSound
```

Прототип функции:

```
int DMusic_Load_MIDI(char *filename);
```

Назначение:

Функция DMusic_Load_MIDI() загружает MIDI-сегмент в память и выделяет запись в массиве midi_ids[]. Функция возвращает идентификатор загруженного MIDI-сегмента или -1 в слу-

чае неуспешного завершения. Полученный идентификатор используется в качестве ссылки при любых обращениях к другим функциям. Вот пример загрузки пары MIDI-файлов:

```
// Загружаем файлы
int explode_id = DMusic_Load_MIDI("explosion.mid");
int weapon_id  = DMusic_Load_MIDI("laser.mid");

// Проверка успешности загрузки
if (explode_id == -1 || weapon_id == -1)
   { /* Ошибка */ }
```

Прототип функции:

```
int DMusic_Delete_MIDI(int id);
```

Назначение:

Функция DMusic_Delete_MIDI() удаляет ранее загруженный MIDI-сегмент из системы. Для этого вам нужно просто передать идентификатор сегмента функции.

```
if (!DMusic_Delete_MIDI(explode_id)||
   !DMusic_Delete_MIDI(weapon_id))
   { /* Ошибка */ }
```

Прототип функции:

```
int DMusic_Delete_All_MIDI(void);
```

Назначение:

Посредством обращения к функции DMusic_Delete_All_MIDI() из системы удаляются все MIDI-сегменты.

```
// Удаление обоих сегментов
if (!DMusic_Delete_ALL_MIDI())
   { /* ошибка */ }
```

Прототип функции:

```
int DMusic_Play (int id);
```

Назначение:

Функция DMusic_Play() воспроизводит MIDI-сегмент. Вам нужно просто передать в функцию идентификатор сегмента, подлежащего воспроизведению.

```
// Загружаем файл
int explode_id = DMusic_Load_MIDI("explosion.mid");

// Воспроизводим его
if (!DMusic_Play ( explode_id ) )
   { /* Ошибка */ }
```

Прототип функции:

```
int DMusic_Stop(int id);
```

Назначение:

Функция DMusic_Stop() останавливает воспроизводящийся в данный момент MIDI-сегмент. Если сегмент уже остановлен, эта функция не оказывает никакого действия.

```
// Остановить звук
if (!DMusic_Stop(weapon_id))
   { /* Ошибка */ }
```

Прототип функции:

```
int DMusic_Status_MIDI(int id);
```

Назначение:

Функция DMusic_Status() проверяет состояние любого MIDI-сегмента, используя для этого его идентификатор. Вот возможные коды состояний:

```
#define MIDI_NULL    0    // MIDI-объект не загружен
#define MIDI_LOADED  1  // MIDI-объект загружен
#define MIDI_PLAYING 2  // MIDI-объект загружен и воспроизводится
#define MIDI_STOPPED 3  // MIDI-объект загружен, но остановлен
```

Вот пример изменения состояния игры после завершения исполнения MIDI-сегмента:

```
// Основной цикл игры
while(1)
{
  if (DMusic_Status(explode_id)== MIDI_STOPPED)
    game_state = GAME_MUSIC_OVER;
} // while
```

На прилагаемом компакт-диске находятся программы DEMO10_5.CPP и DEMO10_6.CPP, демонстрирующие использование рассмотренной библиотеки. Первая из них — это демонстрационная программа DirectMusic, которая позволяет вам выбрать MIDI-файл из меню и воспроизвести его. Вторая — это приложение, работающее в смешанном режиме и использующее одновременно и DirectSound и DirectMusic. Важной деталью второй демонстрационной программы является то, что вначале необходимо инициализировать DirectSound. Звуковая библиотека обнаруживает этот факт и подключается к DirectSound (в противном случае звуковая библиотека должна создавать собственный объект DirectSound).

ВНИМАНИЕ

Обе программы — DEMO10_5.CPP и DEMO10_6.CPP — используют внешний курсор, пиктограмму и ресурсы меню, содержащиеся в файлах DEMO10_5.RC и DEMO10_6.RC соответственно. При компиляции приложений убедитесь в том, что эти файлы включены в проект. Кроме того, для успешного выполнения компиляции не забудьте включить библиотеку T3DLIB3.CPP!

Резюме

В этой главе рассмотрен ряд принципиальных вопросов работы со звуком. Вы узнали о природе звука и музыки, о работе синтезаторов и способах записи звука. Здесь вы познакомились с DirectSound и DirectMusic, создали библиотеку и просмотрели множество демонстрационных примеров. Теперь вы знаете все, что нужно для создания игр, и можете смело переходить к следующей части книги!

ЧАСТЬ III

Программирование игр

Глава 11

Алгоритмы, структуры данных, управление памятью и многопоточность 593

Глава 12

Искусственный интеллект в игровых программах 647

Глава 13

Основы физического моделирования 707

Глава 14

Генерация текста 765

Глава 15

Разработка игры Outpost 801

ГЛАВА 11

Алгоритмы, структуры данных, управление памятью и многопоточность

В этой главе речь пойдет о довольно серьезных вещах, которым обычно уделяется слишком мало внимания в книгах о программировании игр.

- Структуры данных
- Анализ алгоритмов
- Теория оптимизации
- Математические основы
- Программирование на разных языках
- Сохранение игр
- Игры для многих игроков
- Методы многопоточного программирования

Структуры данных

Пожалуй, наиболее часто мне задают вопрос: “Какие структуры данных должны использоваться в играх?” Ответ прост: те, которые обеспечат наиболее быструю и эффективную работу. В большинстве случаев вам не требуется ничего более сложного, чем предлагается в учебнике по информатике. Более того, как правило, чем решение проще, тем оно эффективнее. И еще одно замечание: в наше время скорость важнее, чем память, так что в первую очередь в жертву приносится именно память, а уж потом — скорость работы.

Запомнив сказанное, рассмотрим некоторые наиболее часто используемые структуры данных и выясним, когда и как следует их применять.

Статические структуры и массивы

Среди прочих структур данных наиболее фундаментальной, конечно же, является обычная структура или класс, представляющие единый набор элементов-данных, например:

```
typedef struct PLAYER_TYP
{
  int state;
  int x,y;
  // ...
} PLAYER, *PLAYER_PTR;
PLAYER player_1, player_2;
```

C++

В C++ для создания типа не требуется использовать typedef в определении структуры — тип создается автоматически, когда вы используете ключевое слово struct. Кроме того, в C++ структуры могут содержать методы, а также открытые и закрытые разделы.

В данном случае создаются две статические переменные, представляющие описанную выше структуру данных. Если вам требуется большое количество переменных этого типа, вероятно, будет проще использовать массив

```
PLAYER players[MAX_PLAYERS];
```

В таком случае вы сможете работать со всеми переменными в одном цикле.

"Конечно, это хорошо, но что делать, если заранее не известно, сколько переменных может понадобиться?", — можете возразить мне вы. В таких случаях я обычно оцениваю максимально возможное число элементов, которые могут храниться в массиве. Если это сравнительно небольшое число (скажем, не превышающее 256) и если каждый элемент массива также невелик (скажем, не более тех же 256 байт), то в таком случае я обычно выделяю статический массив и использую счетчик для отслеживания реального количества элементов в массиве.

Такой способ может показаться вам непроизводительной тратой памяти. Что ж, это так, но не забывайте, что предварительно выделенный массив с элементами фиксированного размера проще и быстрее обходится, чем, например, динамически растущий связанный список или какая-то другая динамическая структура. Хочу подчеркнуть: если вы заранее знаете количество элементов и их не так уж много, то лучше заранее выделить для них массив вызовом malloc() или new() в процессе инициализации.

ВНИМАНИЕ

Не увлекайтесь статическими массивами чрезмерно. Допустим, у вас есть структура размером 4 Кбайт и в массиве может находиться от 1 до 256 элементов. В этой ситуации, пожалуй, лучше использовать связанный список или массив динамически изменяемого при необходимости размера, чем сразу же выделять 1 Мбайт памяти в расчете на то, что когда-нибудь вам может понадобиться разместить все 256 элементов.

Связанные списки

Массивы хорошо подходят в качестве простых структур данных, когда количество элементов заранее известно (или может быть оценено) в процессе компиляции или при инициализации программы. Однако если количество данных в процессе работы программы может расти или уменьшаться, то лучше использовать некоторую разновидность связанных списков. На рис. 11.1 изображен стандартный абстрактный связанный список, состоящий

из узлов, каждый из которых содержит информацию и связь (представляющую собой указатель) со следующим узлом в списке.

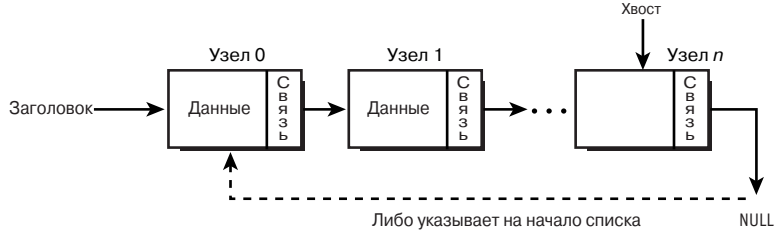


Рис. 11.1. Связанный список

Вы легко можете как вставить новый элемент в любое место в списке, так и удалить элемент из списка. Взгляните на рис. 11.2, чтобы понять, как осуществляется вставка в список нового элемента. Легкость вставки и удаления узлов в процессе работы программы делает связанные списки весьма привлекательными для использования в играх.

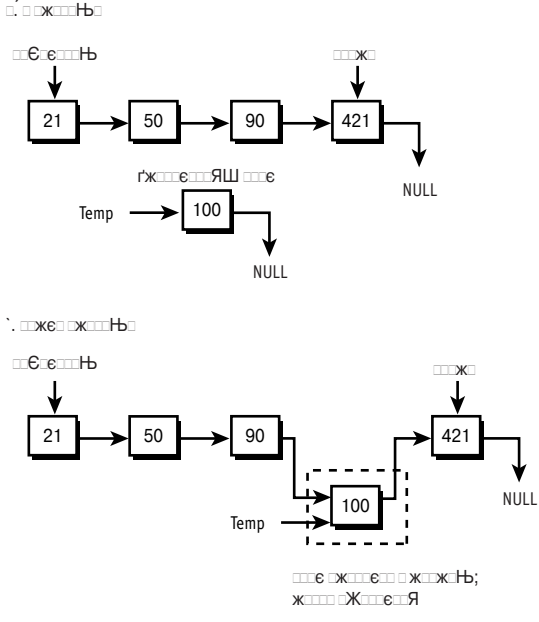


Рис. 11.2. Вставка в связанный список

Единственный неприятный момент при работе со связанными списками состоит в том, что при поиске определенного узла требуется последовательный обход узла за узлом (если только нет еще одной структуры данных, облегчающей выполнение поиска). Например, обратиться к 15-му элементу массива очень просто — players[15]. Но если players — связанный список, то вам потребуется алгоритм прохода по списку для обнаружения нужного узла. Это означает, что поиск узла в связанном списке требует значительного количества итераций, в худшем случае равного длине списка, а в среднем — половине длины. Таким образом, требуется *порядка n операций при поиске узла в списке из n элементов*, или в краткой записи — $O(n)$ операций. Конечно, при наличии соответст-

вующей вторичной структуры данных поиск в связанном списке можно сделать практически таким же быстрым, как и в простом массиве.

Создание связанного списка

Рассмотрим, как создается связанный список, а также операции по добавлению в него элемента, удалению и поиску элемента с данным ключом. В качестве примера будем работать со списком, узлы которого представляют собой следующие структуры:

```
typedef struct NODE_TYP
{
  int id;              // Идентификатор объекта
  int age;             // Возраст персонажа
  char name[32];       // Имя персонажа
  NODE_TYP *next;    // Указатель на следующий узел списка
  // ...могут иметься и другие поля...
} NODE, *NODE_PTR;
```

Все, что нужно для создания списка, — это его *заголовок*, т.е. указатель на первый элемент списка. Если список пуск, значение этого указателя равно NULL:

```
NODE_PTR head = NULL;
```

Обход связанного списка

Это самая простая операция со связанным списком. Для этого требуется совсем немного.

1. Начать с заголовка списка.
2. Посетить узел.
3. Найти указатель на следующий за ним узел.
4. Если он не равен NULL, перейти к п. 2.

Вот соответствующий код:

```
void Traverse_List(NODE_PTR head)
{
  // Обходим список и выводим информацию о каждом узле

  if (head == NULL)
  {
    printf("Список пуст!\n");
    return;
  } // if

  // Обходим узлы
  while(head != NULL)
  {
    printf("Узел:  id = %d\n"
         "      age = %d\n"
         "     name = %s\n",
         head->id, head->age, head->name);

    head = head->next;  // Переход к следующему узлу
  } // while
} // Traverse_List
```

Теперь перейдем к более сложной задаче — добавлению узлов в список.

Добавление узла в связанный список

Первый шаг при добавлении элемента в связанный список — это его создание. Возможны два пути: передать в функцию данные и позволить ей создать узел с этими данными либо самостоятельно создать узел и передать его соответствующей функции для добавления в список.

Кроме того, имеется ряд различных вариантов добавления узла в связанный список. Простейшие из них — добавление элемента в начало или в конец списка. Если вас не интересует упорядоченность создаваемого списка, то эти методы вполне приемлемы, но, если элементы в списке должны быть отсортированы, придется прибегнуть к более интеллектуальным алгоритмам — по сути, к алгоритму сортировки вставкой.

Для простоты я воспользуюсь вставкой в конец списка (что, кстати, все же сложнее, чем вставка в его начало), хотя во вставке с сортировкой тоже нет ничего сложного: вы просто должны пройти по списку, найти место, где должен быть вставлен новый элемент, и вставить его.

```
NODE_PTR Insert_Node(NODE_PTR *head, int id,
            int age, char * name)
{
  // Вставка узла в конец списка
  NODE_PTR new_node = NULL, curr = *head;

  // Создание нового узла
  // В C++ здесь используется оператор new
  new_node = (NODE_PTR)malloc(sizeof(NODE));

  // Заполняем поля
  new_node->id  = id;
  new_node->age = age;
  strcpy(new_node->name,name); // Строку следует копировать
  new_node->next = NULL;          // Лучше это делать всегда;
                                  // в данном случае наш узел будет
                                  // последним в списке, так что это
                                  // просто необходимо
  // Поиск конца списка и вставка
  if (curr == NULL) // Список пуст
  {
    (*head) = new_node;
    return(new_node);
  } // if
  else
  {
    while(curr->next != NULL) curr = curr->next;
    curr->next = new_node;
    return(new_node);
  } // else
} // Insert_Node
```

Обратите внимание на использование в качестве параметра указателя на заголовок списка, который необходим для корректного обновления его значения при вставке в пустой список.

Удаление узла

Вряд ли эту задачу можно назвать сложной. Обычно требуется удалить конкретный узел, и вы должны рассмотреть варианты, когда этот узел находится в начале, конце или середине списка. Если вы аккуратны, удаление узла не составит проблемы.

Алгоритм удаления узла должен обеспечить его поиск, удалить узел и освободить занимаемую им память. Кроме того, алгоритм должен обеспечить корректное изменение значений указателей. Принцип удаления узла из списка показан на рис. 11.3, а ниже приведен код удаления из списка узла с определенным идентификатором.

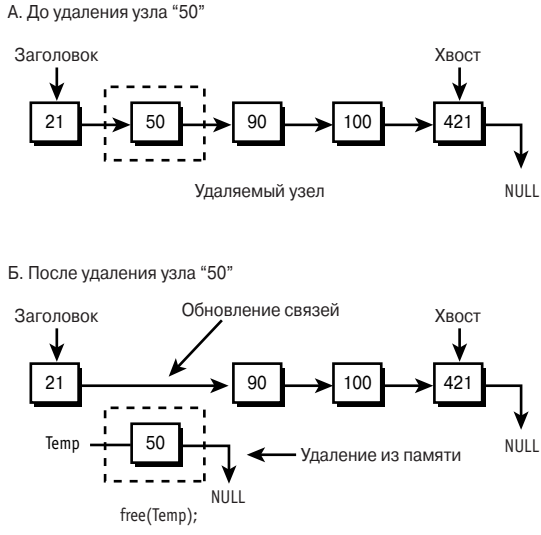


Рис. 11.3. Удаление узла из связанного списка

```
int Delete_Node(NODE_PTR * head, int id)
{
  NODE_PTR curr_ptr = *head,
       prev_ptr = *head;

  if (curr_ptr == NULL) // Список пуст
    return(-1);

  // Обход списка и поиск удаляемого узла
  while(curr_ptr && curr_ptr->id != id)
  {
    // Сохраняем положение в списке
    prev_ptr = curr_ptr;
    curr_ptr = curr_ptr->next;
  } // while

  if (curr_ptr == NULL)          // Узел не найден
    return(-1);

  if (curr_ptr == *head)         // Узел в начале списка
  {
    *head = (*head)->next;   // Обновляем заголовок
  } //if
  else
  {
    // Обновляем указатель
```

```
    prev_ptr->next = curr_ptr->next;
  } //else
  free(curr_ptr);      // Освобождаем память
  return(id);
}
```

В этом коде следует обратить внимание на передачу функции адреса заголовка списка и на наличие дополнительной переменной prev_ptr, необходимость в которой объясняется тем, что при прохождении определенного узла списка нет возможности вновь вернуться к нему и поэтому требуется запоминать последний пройденный узел для корректного обновления указателей. Эта проблема легко решается использованием *двухсвязного списка*, показанного на рис. 11.4.

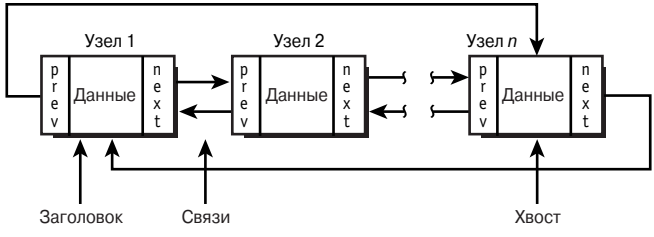


Рис. 11.4. Двухсвязный список

Двухсвязный список отличается наличием двух указателей — на следующий узел в списке и на предыдущий, что существенно упрощает операции вставки и удаления. При этом в структуру узла вносится дополнительный указатель.

```
typedef struct NODE_TYP
{
  int id;              // Идентификатор объекта
  int age;             // Возраст персонажа
  char name[32];       // Имя персонажа
  NODE_TYP *next;      // Указатель на следующий узел списка
  NODE_TYP *prev;      // Указатель на предыдущий узел списка
  // ...могут иметься и другие поля...
} NODE, *NODE_PTR;
```

При работе с двухсвязными списками вы всегда можете перейти к следующему или предыдущему узлу списка.

На прилагаемом компакт-диске находится демонстрационная программа DEMO11_1.CPP, в которой реализованы добавление узла в простой связанный список, удаление узла и обход списка. Главное отличие исходного текста демонстрационной программы от приведенных в книге фрагментов заключается в том, что для простоты в программе для хранения заголовка списка используется глобальная переменная, а следовательно, заголовок списка в качестве параметра в функции вставки и удаления не передается.

НА ЗАМЕТКУ

Программа DEMO11_1.CPP представляет собой консольное приложение, так что при ее компиляции установите соответствующие опции компилятора. Кроме того, она не использует DirectX, так что никакие .LIB-файлы DirectX для компиляции данного приложения не нужны.

Анализ алгоритмов

Разработка и анализ алгоритмов, как правило, рассматриваются только в серьезных научных трудах или по крайней мере в учебниках для старшекурсников, изучающих информатику. В нашей книге только слегка затрагивается эта тема, чтобы дать общее представление, которое поможет вам при разработке собственных алгоритмов.

Хороший алгоритм — гораздо более важная для эффективной работы вещь, чем все языки ассемблера и оптимизации, вместе взятые. Одно лишь переупорядочение данных способно в несколько раз уменьшить время, затрачиваемое на поиск. Выбор хорошего алгоритма и подходящей для работы с ним структуры данных — самое главное для повышения эффективности разрабатываемой программы.

Например, при использовании обычного массива вы никогда не сможете достичь лучшего результата, чем линейная зависимость времени поиска от количества элементов (если, конечно, не используете при этом дополнительные структуры данных). Однако стоит лишь использовать массив с отсортированными данными, как ситуация коренным образом меняется и вы получаете логарифмическую зависимость времени поиска от количества данных.

Первым шагом в разработке хорошего алгоритма является его *асимптотический анализ*, в который я не стану углубляться, а познакомлю вас с ним, как говорится, "на пальцах".

Основная идея анализа алгоритмов такова: вычислить, сколько раз требуется выполнить основной цикл алгоритма при наличии n элементов, каким бы ни было это значение n. Конечно, важно также определить время выполнения каждой операции, дополнительные расходы на настройку для работы и т.п., но все же самое главное — определение количества итераций цикла. Рассмотрим два примера.

```
for(int index = 0; index < n; ++index)
{
  // Выполняем некоторые действия, 50 тактов
} //for index
```

В нашем случае выполняются n итераций цикла, следовательно, время выполнения порядка n, или $O(n)$. Эта запись называется *записью с большим O* и представляет собой грубую оценку времени исполнения сверху. Если вы хотите быть более точны, то можно заметить, что внутри цикла выполняются некоторые действия в количестве 50 тактов. Таким образом, более точная оценка дает нам $50*n$ тактов. Верно? Нет, не верно! Если уж вы хотите считать все такты процессора, то следует учитывать инициализацию счетчика цикла, сравнения, увеличения счетчика и переходы при каждой итерации. В результате получится что-то вроде

$$Cycles_{init}+(50+Cycles_{inc}+Cycles_{comp}+Cycles_{jump})*n$$

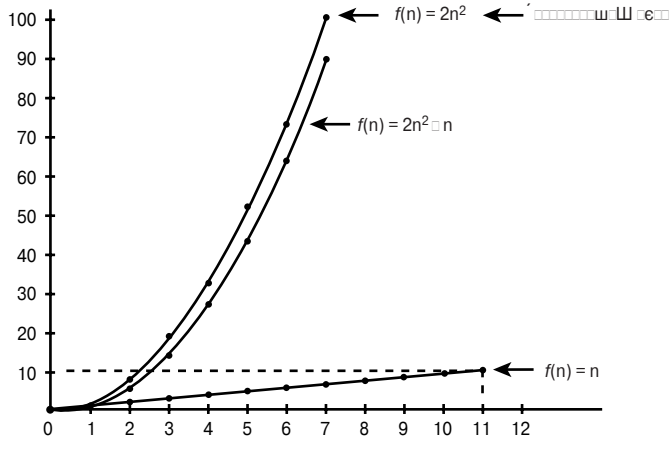
Таким образом, вклад дополнительных действий становится сравним со вкладом основных вычислений. Например, многие программисты реализуют вывод точки в виде функции, а не макроса или встроенного кода. Поскольку вывод точки — очень простая операция, затраты времени на вызов функции могут превышать затраты на вывод точки.

Вот еще один пример оценки времени работы алгоритма:

```
// Внешний цикл
for(i=0; i<n; ++i)
{
  // Внутренний цикл
  for(j=1; j<2*n; ++j)
  {
```

```
    // Выполнение некоторых действий
  } // for j
} // for i
```

Будем считать, что время выполнения действий во внутреннем цикле существенно больше времени, затрачиваемого на поддержку работы циклов. Нас интересует только одно: сколько раз будет выполнен цикл. Внешний цикл выполняется n раз, и при каждом его выполнении внутренний цикл выполняется $2n$-1 раз. Следовательно, общее количество выполнений внутреннего цикла составляет $n(2n-1) = 2n^2 - n$ раз. Это выражение состоит из двух членов. Член $2n^2$ доминирующий, и при больших значениях n вторым членом можно пренебречь (рис. 11.5).



n□	n□	2n²□	2n² □ n
0□	0□	0□	0
1□	1□	2□	1
2□	2□	8□	6
3□	3□	18□	15
4□	4□	32□	28
5□	5□	50□	45
6□	6□	72□	66
7□	7□	98□	91
8□	8□	128□	120
9□	9□	102□	153
10□	10□	200□	190

Рис. 11.5. Скорости роста членов функции $2n^2 - n$

При малых n, например при $n = 2$, второй член вносит существенный вклад в результат: $2 \cdot 2^2 - 2 = 6$, а n составляет 25% от $2n^2$. Однако, например, при $n = 1000$ картина существенно изменяется и значение члена n составляет всего 0.05% от $2n^2$. Так что с ростом n вторым членом можно пренебречь, и считать общее количество циклов равным $2n^2$, а алгоритм — имеющим время работы $O(n^2)$. Это очень плохой алгоритм, так что необходимо искать другой способ решения задачи.

На этом завершим экскурс в асимптотический анализ. Главный вывод заключается в том, что вы должны быть способны хотя бы грубо оценить время работы ваших циклов и определить качество разрабатываемых алгоритмов. Это поможет определить, что именно в программе следует переделывать в первую очередь.

Рекурсия

Следующая тема, которую я хочу здесь затронуть, — рекурсия. Это метод решения задач по индукции, основанный на том, что многие задачи сводятся к более простым задачам того же вида и так продолжается до тех пор, пока в некоторый момент задача не оказывается решенной.

В программировании рекурсивные алгоритмы обычно используются для поиска, сортировки и некоторых математических операций. Вы пишете функцию, которая для решения задачи в какой-то момент вызывает сама себя. Звучит странно? Нисколько. Дело в том, что при таком вызове в стеке создается новый набор локальных переменных, так что, по сути, можно считать, что вызывается другая функция. Единственное, о чем следует побеспокоиться при использовании рекурсии, — чтобы вызовы функцией самой себя не привели к переполнению стека и чтобы в процессе вычислений обязательно достигалось условие завершения, т.е. чтобы выполнение такой функции гарантированно завершалось.

Рассмотрим стандартный пример — вычисление *факториала*. Факториал числа *n* (записывается как *n*!) означает следующее:

$$n! = \prod_{j=1}^{n} j = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 .$$

Кроме того, принимаем, что 0! = 1. Таким образом, например, 5! = 5*4*3*2*1 = 120.

Написать код для вычисления факториала несложно:

```
int Factorial(int n)
{
  int sum = 1;  // Переменная для хранения результата
  for(;n >= 1; --n) sum *= n;
  return(sum);
} // Factorial
```

Выглядит просто. Если вы передадите в функцию значение 0, то получите значение 1. При передаче значения 3 будет выполнена такая последовательность вычислений:

```
sum = sum * 3 = 1 * 3 = 3
sum = sum * 2 = 3 * 2 = 6
sum = sum * 1 = 6 * 1 = 6
```

Все правильно: 3! = 6.

А вот как выглядит рекурсивное вычисление факториала:

```
int Factorial_Rec(int n)
{
  return ((n <= 1) ? 1 : n*Factorial_Rec(n-1));
} // Factorial_Rec
```

И попробуйте только сказать, что это некрасиво!

Посмотрим теперь, что произойдет, если *n* равно 0 или 1. При этом условие n <= 1 оказывается истинным и функция возвращает значение 1. Но самое интересное начинается, когда функции передается значение, большее 1. В этом случае возвращается увеличенное в *n* раз значение функции, которой передается параметр (*n*-1). Это и есть рекурсия — вызов функцией самой себя. При этом текущие переменные функции сохраняются в стеке, а вновь вызываемая функция получает новый набор переменных. Код первого оператора return не завершается до тех пор, пока рекурсивно вызванная функция не вернет вычисленное значение (а ее оператор return, в свою очередь, требует завершения вычисления рекурсивно вызванной из него функции — и так до достижения условия завершения рекурсии).

Поясним это на примере вычисления значения 3!.

1. Изначально вызываем функцию Factorial_Rec(3). Этот вызов сводится к оператору return(3*Factorial_Rec(2)).
2. Второй вызов — Factorial_Rec(2) — сводится к оператору return(2*Factorial_Rec(1)).
3. В третьем вызове — Factorial_Rec(1) — достигается условие завершения и без дальнейших рекурсивных вызовов функция возвращает значение 1.
4. Это значение подставляется во второй вызов, который возвращает return(2*Factorial_Rec(1)), т.е. значение 2.
5. Возвращенное значение 2 подставляется в оператор return(3*Factorial_Rec(2)) начального вызова функции, в результате будет получено значение3!, равное 6.

Совершенно естественным будет вопрос: так какой же метод лучше — рекурсивный или нет? Очевидно, что нерекурсивный метод работает быстрее хотя бы в силу того, что в нем отсутствуют накладные расходы на вызовы функций, но рекурсия более элегантна и лучше соответствует поставленной задаче (например, в случае вычисления факториала его рекурсивность проявляется в очевидном тождестве $n! \equiv n \cdot (n-1)!$). Многие задачи рекурсивны по своей природе, и зачастую разработка нерекурсивного алгоритма для их решения сводится к имитации рекурсии вручную! Поэтому смело используйте рекурсию там, где она лежит в самой природе вещей и упрощает решение задачи. В противном случае лучше использовать обычный нерекурсивный код.

На прилагаемом компакт-диске находится демонстрационная программа DEMO11_2.CPP, которая реализует алгоритм вычисления факториала.

Попробуйте реализовать рекурсивный алгоритм вычисления чисел Фибоначчи, которые имеют следующее определение: $F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$. Таким образом, начало последовательности чисел Фибоначчи выглядит следующим образом: 0, 1, 1, 2, 3, 5, 8, 13...

Деревья

Следующий класс структур данных, с которым я хочу вас познакомить, — это деревья. Работа с ними осуществляется преимущественно с использованием рекурсивных алгоритмов (именно поэтому деревья рассматриваются после изучения рекурсии). На рис. 11.6 показаны различные древообразные структуры данных.

Деревья помогают хранить большое количество данных и упрощают поиск. Наиболее распространены бинарные деревья (используются также названия B-tree и BST (Binary Search Tree — дерево бинарного поиска)), которые "вырастают" из одного корня и состоят из множества узлов, причем каждый узел имеет не более двух дочерних узлов (отсюда и происходит название *бинарные* деревья). Кроме того, можно говорить о *порядке*, или количестве уровней дерева, которое, по сути, означает максимальную длину пути от корня к узлу дерева (деревья различного порядка показаны на рис. 11.7).

Бинарные деревья позволяют осуществлять очень быстрый поиск данных. Большинство бинарных деревьев используют для доступа к данным один ключ поиска. Чтобы облегчить понимание того, как выполняется поиск, рассмотрим конкретный пример. Пусть требуется создать бинарное дерево, содержащее записи об объектах

игры, каждая с определенным набором свойств. В качестве ключа можно использовать, например, время создания объекта. Вот структура данных, которая позволяет хранить отдельный узел с информацией об объекте:

```
typedef struct TNODE_TYP
{
  int     age;          // Возраст объекта
  char    name[32];     // Имя объекта
  NODE_TYP *right;      // Указатель на правый дочерний узел
  NODE_TYP *left;       // Указатель на левый дочерний узел
} TNODE, *TNODE_PTR;
```

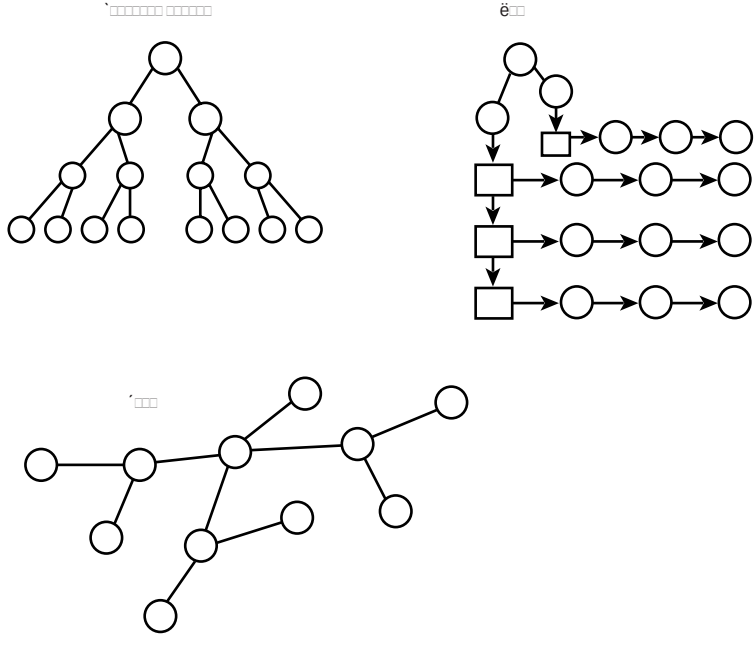


Рис. 11.6. Топологии деревьев

Обратите внимание на схожесть узла дерева и узла связанного списка. Главное отличие заключается в способе построения и использования дерева. Вернемся к нашему примеру: пусть у нас имеются объекты с возрастом 5, 23, 3, 12 и 10. На рис. 11.8 изображены два различных бинарных дерева, содержащих эти данные. Вы можете создавать самые разные деревья, в зависимости от того, какой алгоритм упорядочения вы используете при вставке узлов в дерево.

НА ЗАМЕТКУ Понятно, что данные, которые вы используете, могут быть любыми, а не только приведенными в данном примере.

Здесь я использовал следующее соглашение: любой правый дочерний узел содержит ключ, значение которого больше или равно значению ключа родительского узла; значение же ключа левого узла соответственно меньше значения ключа родительского узла. Вы же можете использовать и другие правила.

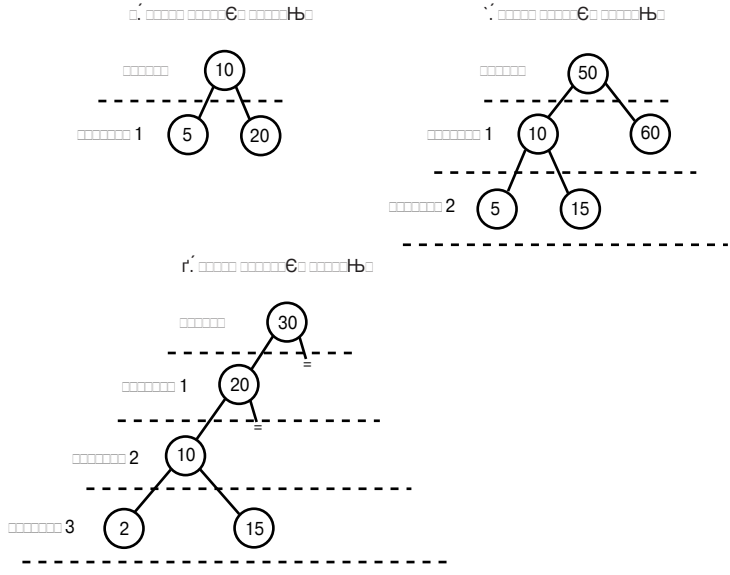
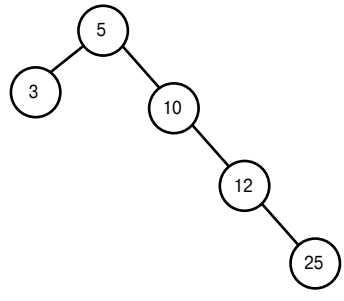


Рис. 11.7. Бинарные деревья и их порядки



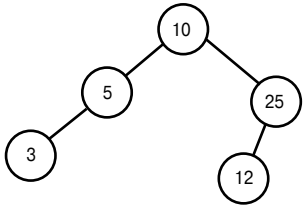


Рис. 11.8. Бинарные деревья, представляющие множество {5, 23, 3, 12, 10}

Бинарные деревья могут хранить огромные количества данных, и поиск среди этих данных может выполняться очень быстро. Например, при наличии миллиона узлов при

бинарном поиске вам придется выполнить всего лишь около 20 сравнений. Причина в том, что при каждой итерации поиска в хорошо сбалансированном дереве вы удаляете из зоны поиска примерно половину содержащихся в ней узлов. Одним словом, при наличии n узлов вам требуется $\log_2 n$ сравнений; соответственно время работы алгоритма бинарного поиска — $O(\log n)$[1].

НА ЗАМЕТКУ

Сказанное выше справедливо для поиска в сбалансированных деревьях, т.е. деревьях, имеющих на каждом уровне одинаковое количество правых и левых дочерних узлов. Несбалансированное дерево в предельном случае вырождается в связанный список, поиск в котором, как вы уже знаете, имеет время работы $O(n)$.

Еще одно полезное свойство бинарных деревьев состоит в том, что если вы возьмете любую его ветвь (поддерево), то можете работать с ней отдельно — она сохраняет все свойства бинарного дерева. Следовательно, если вы знаете, где именно вести поиск, то можно ограничиться поиском в конкретном поддереве. Таким образом, для сокращения поиска можно создать деревья деревьев или индексные таблицы, которые содержат поддеревья, и вам не потребуется работать со всем деревом целиком. В случае моделирования трехмерного мира это очень важно. У вас может быть огромное бинарное дерево, описывающее весь мир целиком, и сотни поддеревьев, представляющих отдельные помещения. При этом у вас может быть еще одно дерево, которое представляет собой пространственно отсортированный список указателей на эти поддеревья, как показано на рис. 11.9. Эту тему мы еще рассмотрим в книге более подробно.

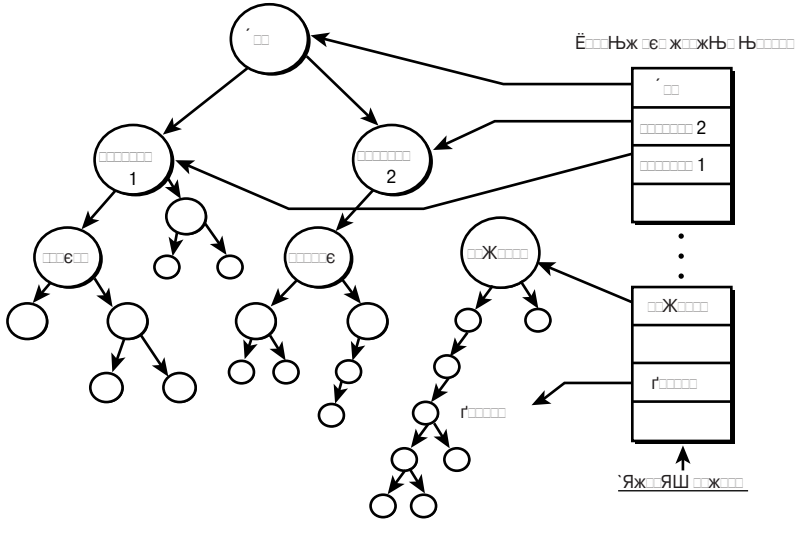


Рис. 11.9. Использование вторичной индексной таблицы для работы с бинарным деревом

И наконец, рассмотрим, когда же следует использовать бинарные деревья. Лично я предлагаю использовать их тогда, когда задача или данные имеют выраженный "древообразный" вид. Если вы делаете набросок задачи на бумаге и видите, что на ри-

[1] Пропущенное основание логарифма не опечатка; вспомните, что логарифмы по разным основаниям отличаются постоянным множителем: $\log_y x = \log_y z \cdot \log_z x$. — *Прим. ред.*

сунке появляются ветви влево и вправо, определенно стоит подумать об использовании деревьев при решении данной задачи.

Построение бинарных деревьев

Эта тема достаточно сложна в силу рекурсивной природы всех алгоритмов, работающих с бинарными деревьями. Мы рассмотрим некоторые из них, напишем соответствующие фрагменты кода и в завершение обратимся к демонстрационной программе.

Так же, как и при работе со связанными списками, имеется ряд способов начать построение бинарного дерева. Можно начать как с реального корня дерева, так и с пустого узла, призванного играть роль корня. Я предпочитаю использовать только реальные узлы, и, соответственно, указатель на корневой узел пустого дерева имеет значениеNULL.

```
TNODE_PTR root = NULL;
```

Для того чтобы вставлять узел в бинарное дерево, следует решить, что будет использоваться в качестве ключа вставки. В нашем случае можно просто использовать возраст или имя объекта. Хотя в качестве ключа используется возраст, применение в качестве ключа имени также довольно просто: в этом случае для упорядочения используется лексикографическая функция сравнения, такая, как strcmp().

Вот функция для вставки нового узла в дерево:

```
TNODE_PTR root = NULL; // Корень дерева

TNODE_PTR BST_Insert_Node(TNODE_PTR root, int id,
                  int age, char *name)
{
// Проверка на пустоту дерева
if (root==NULL)
  {
  // Вставка узла в корень
  root       = new(TNODE);
  root->id    = id;
  root->age   = age;
  strcpy(root->name,name);

  // Установка указателей на дочерние узлы
  root->right  = NULL;
  root->left   = NULL;

  printf("\nCreating tree");

  } // if

// Если узел имеется, идем влево или вправо
else
if (age >= root->age)
  {
  printf("\nTraversing right...");
  // Вставка в правую ветвь

  if (root->right)
    BST_Insert_Node(root->right, id, age, name);
  else
```

```
    {
    // Вставка узла справа
    TNODE_PTR node   = new(TNODE);
    node->id     = id;
    node->age    = age;
    strcpy(node->name,name);

    // Установка указателей на дочерние узлы
    node->left   = NULL;
    node->right  = NULL;

    // Устанавливаем значение правого
    // указателя "корневого" узла
    root->right = node;

    printf("\nInserting right.");

    } // else

  } // if
else // age < root->age
  {
  printf("\nTraversing left...");
  // Вставка в левую ветвь

  if (root->left)
    BST_Insert_Node(root->left, id, age, name);
  else
    {
    // Вставка узла слева
    TNODE_PTR node   = new(TNODE);
    node->id     = id;
    node->age    = age;
    strcpy(node->name,name);

    // Установка указателей на дочерние узлы
    node->left   = NULL;
    node->right  = NULL;

    // Устанавливаем значение левого
    // указателя "корневого" узла
    root->left = node;

    printf("\nInserting left.");
    } // else

} // else

// Возвращаем корень дерева
return(root);

} // BST_Insert_Node
```

Сначала необходимо проверить, не пусто ли данное дерево, и если это так, то создать его корень. Следовательно, первый элемент, вставляемый в бинарное дерево, должен находиться в средине пространства поиска с тем, чтобы дерево было более-менее сбалансировано. Если же дерево содержит более одного узла, вы обходите его, переходя к левым или правым дочерним узлам, в зависимости от значения ключа вставляемого узла. Найдя соответствующую ветвь, вы используете рекурсивный вызов функции `BST_Insert_Node()`. На рис. 11.10 показан пример вставки в дерево узла `"Jim"`.

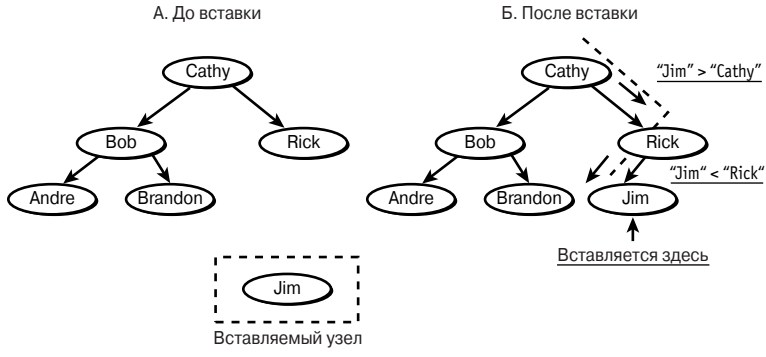


Рис. 11.10. Вставка в бинарное дерево

Время, необходимое для вставки узла в бинарное дерево, такое же, как и для поиска в нем, и равно в среднем $O(\log n)$, а в худшем случае — $O(n)$ (если ключи поступают в дерево упорядоченно).

Поиск в бинарном дереве

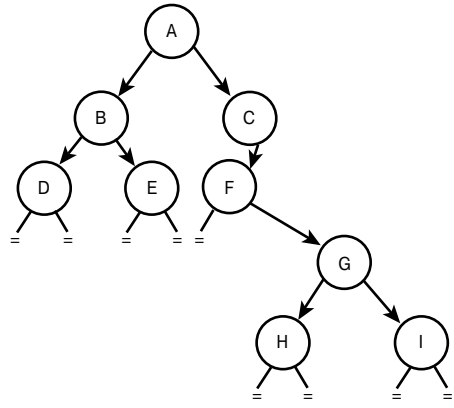
После того как бинарное дерево создано, поиск в нем — очень простая задача. Однако и здесь есть свои тонкости. Имеется три пути обхода узлов в бинарном дереве.

- **Прямой порядок**. Посещаем узел, осуществляем обход в прямом порядке его левого поддерева, а затем его правого поддерева.
- **Внутренний порядок**. Обходим во внутреннем порядке левое поддерево, посещаем корневой узел, а затем обходим во внутреннем порядке правое дерево.
- **Обратный порядок**. Обходим в обратном порядке левое поддерево, затем правое поддерево, после чего посещаем корневой узел.

НА ЗАМЕТКУ Выбор правых и левых узлов в данном случае произволен — важен лишь порядок их обхода.

Взгляните на рис. 11.11 — на нем показано дерево и три способа его обхода.

Теперь разработаем очень простой рекурсивный алгоритм выполнения обхода дерева. Конечно, цель обхода состоит в том, чтобы найти нечто и вернуть найденное как результат работы функции, но все, что делает приведенная далее функция, — это обход дерева (конечно, вы можете добавить к ней свой код, который, например, прекратит обход при достижении необходимого узла).



□□□□□Ш □□□□□Нь: ABDECFGHI
Г□□□□□□□Ш □□□□□Нь: DBEAFHGIC
□Ж□□□ЯШ □□□□□Нь: DEBHIGFCA

Рис. 11.11. Способы обхода узлов бинарного дерева

Обход дерева во внутреннем порядке:

```
void BST_Inorder_Search(TNODE_PTR root)
{
// Проверка на пустоту дерева
if (!root)
   return;

// Обход левого дерева
BST_Inorder_Search(root->left);

// Посещение корневого узла
printf("\nname: %s, age: %d", root->name, root->age);

// Обход правого дерева
BST_Inorder_Search(root->right);

} // BST_Inorder_Search
```

Обход дерева в прямом порядке:

```
void BST_Preorder_Search(TNODE_PTR root)
{
// Проверка на пустоту дерева
if (!root)
  return;

// Посещение корневого узла
printf("\nname: %s, age: %d", root->name, root->age);

// Обход левого дерева
BST_Preorder_Search(root->left);

// Обход правого дерева
```

```
BST_Preorder_Search(root->right);

} // BST_Preorder_Search
```

Обход дерева в обратном порядке:

```
void BST_Postorder_Search(TNODE_PTR root)
{
// Проверка на пустоту дерева
if (!root)
  return;

// Обход левого дерева
BST_Postorder_Search(root->left);

// Обход правого дерева
BST_Postorder_Search(root->right);

// Посещение корневого узла
printf("\nname: %s, age: %d", root->name, root->age);

} // BST_Postorder_Search
```

Обратите внимание, что я не рассказываю о том, как удалить узел из дерева. Это очень сложный вопрос, так как при удалении имеется риск удалить не только узел, но и поддерево, корневым узлом которого является удаляемый узел. Поэтому удаление узла из дерева остается вам в качестве домашнего задания, в этом вам поможет книга Седжвика (Sedgewick) *Algorithms in C++*.

И наконец, в качестве примера работы с бинарными деревьями вы можете рассмотреть имеющуюся на прилагаемом компакт-диске демонстрационную программу DEMO11_3.CPP. Это такое же консольное приложение, как и предыдущая демонстрационная программа, так что компилировать ее следует с соответствующими ключами компиляции.

Теория оптимизации

По сути, игры — это класс приложений, наиболее требовательных к производительности. Если учесть ограничения, накладываемые аппаратным обеспечением, и желание программистов добавить в игру новые объекты, звуковые и видеоэффекты, усовершенствовать систему искусственного интеллекта, то становится понятно, насколько важна оптимизация при написании видеоигр.

В этом разделе я хочу рассказать только о некоторых методах оптимизации, чтобы вы имели представление о ней и могли самостоятельно изучить эту тему более глубоко с помощью следующих книг: Rick Booth. *Inner Loops*; Mike Abrash. *Zen of Code Optimization*; Mike Schmit. *Pentium Processor Optimization*.

Работайте головой

Первым требованием для написания оптимизированного кода является знание компилятора, используемых типов данных и представления о том, как ваш компилятор C/C++ преобразует код на языке высокого уровня в машинный код. Основная идея — использовать в играх простой код, работающий с простыми структурами данных. Чем более сложный и вычурный код вы используете, тем сложнее компилятору преобразовать его в машинный код и тем медленнее он будет выполняться (в большинстве случаев).

Приведем основные правила, которых желательно придерживаться при написании эффективного кода игровых программ.

- Используйте по возможности 32-битовые данные; 8-битовые данные экономят память, но процессоры Intel оптимизированы для доступа к 32-битовым данным.
- Используйте ключевое слово inline при описании часто вызываемых функций малого размера.
- Не бойтесь использовать глобальные переменные.
- Избегайте использования чисел с плавающей точкой, поскольку операции с целыми числами в основном выполняются быстрее.
- Выравнивайте все структуры данных на границу в 32 байта. Вы можете делать это с помощью директив компилятора или соответствующих директив #pragma в коде программы.
- Никогда не передавайте в функцию данные сложных типов по значению. Используйте вместо этого передачу по указателю или по ссылке.
- Не используйте в коде ключевое слово register. Хотя Microsoft и утверждает, что это делает циклы более быстрыми, при этом в распоряжении компилятора оказывается меньше регистров, в результате чего он генерирует код пониженного качества.
- Если вы программируете на C++, использование классов и виртуальных функций вполне допустимо; не стоит только злоупотреблять наследованием.
- Процессоры класса Pentium используют внутренний кэш кода и данных. Не забывайте об этом и постарайтесь, чтобы ваши функции были относительно невелики и могли полностью размещаться в кэше (16–32 Кбайт). Кроме того, при хранении данных старайтесь записывать их на старое место; это минимизирует обращение к основной памяти и вторичному кэшу.
- Не забывайте, что процессоры класса Pentium представляют собой RISC-процессоры, которые предпочитают короткие команды и позволяют выполнять две или более команды несколькими исполняющими модулями. Не используйте в своих программах огромные выражения; лучше разбивать код на небольшие операторы, даже если все необходимые действия можно записать при помощи одного длинного выражения.

Математические уловки

Поскольку игры требуют огромного количества вычислений, немаловажно знание того, как эффективнее вычислить те или иные математические функции. В этой области также существуют методы повышения эффективности.

- В выражениях всегда используйте числа одного типа: целые с целыми, числа с плавающей точкой — с числами с плавающей точкой. Выполнение преобразований снижает общую производительность вычислений.
- Операция умножения целых чисел на степень двойки эффективно реализуется операцией сдвига влево (а деления — сдвига вправо). Эти же операции следует по возможности использовать и при множителях, отличных от степени двойки, например умножение на 640, которое представляет собой сумму 512 и 128, можно реализовать как два сдвига и суммирование: (n<<7)+(n<<9).[2]

[2] Современные компиляторы выполняют такую оптимизацию самостоятельно. — *Прим. ред.*

- При работе с матрицами рассмотрите возможность использования свойств конкретных матриц, например разреженность (когда большинство элементов матрицы равны 0), для выбора эффективного алгоритма вычислений.
- При работе с константами убедитесь, что они имеют корректный тип и компилятору не придется выполнять преобразования из одного типа в другой. Наилучший способ указания типа константы — ее объявление в стиле C++ с использованием ключевого слова const, например:

  ```
  const double f = 12.45;
  ```

- По возможности избегайте вычисления квадратных корней, тригонометрических и других сложных математических функций. Обычно всегда имеется более простой путь получения тех же результатов, например за счет небольшого снижения точности или использования аппроксимации. Если же такое снижение точности неприемлемо, можно воспользоваться поиском в таблице, который рассматривается несколько позже.
- При необходимости обнуления большого массива можно воспользоваться функцией memset():

  ```
  memset((void*)double_array,0,sizeof(double)*num_elements);
  ```

  Этот метод работает в силу использования IEEE стандарта кодирования чисел с плавающей запятой.
- При вычислении математических выражений постарайтесь предварительно максимально их упростить. Например, выражение $\left((nf+2n)/n\right)$ можно вычислить как $\left(f+2\right)$, сократив и числитель и знаменатель на n. Понятно, что в реальных задачах упрощаемые выражения не будут такими простыми.
- Если вы вычислили некоторое сложное математическое выражение и знаете, что оно понадобится вам несколькими строками позже, сохраните его в переменной, чтобы не вычислять заново, например:

  ```
  // Данная величина используется в нескольких выражениях
  double n_squared = n*n;

  pitch = 34.5*n_squares + 100*rate;
  magnitude = n_squared / length;
  ```

- Убедитесь, что установлены опции компилятора, обеспечивающие генерацию кода, работающего с сопроцессором, а также оптимизацию времени выполнения программы (а не ее размера).

Математические вычисления с фиксированной точкой

Несколько лет назад трехмерные игры использовали математические вычисления с фиксированной точкой, что было связано с более медленными вычислениями с плавающей точкой по сравнению с вычислениями с целыми числами. Наличие современных процессоров типа Pentium II, III снизило остроту данной проблемы.

Тем не менее зачастую преобразование чисел с плавающей точкой в целые остается слишком медленным, и использование операций с числами с фиксированной точкой во внутренних циклах может оказаться неплохой идеей. Такие операции выполняются на ранних процессорах типа Pentium быстрее, чем операции с числами с плавающей точкой, а преобразование к целым числам сводится к использованию целой части такого числа.

Конечно, использование чисел с плавающей точкой во всех вычислениях проще и при современном аппаратном вычислении вполне приемлемо. Тем не менее никакие знания не бывают лишними, и я бы хотел познакомить вас с вычислениями с фиксированной точкой. Лично мне наиболее правильным кажется такой путь: применить оба варианта — как с плавающей, так и с фиксированной точкой — и выяснить, какой метод работает быстрее. Очевидно, что при использовании аппаратного ускорения все это становится излишним и следует обращаться только к числам с плавающей точкой.

А теперь с учетом сказанного выше рассмотрим представление чисел с фиксированной точкой.

Числа с фиксированной точкой

Вся математика с фиксированной точкой базируется на масштабируемых целых числах. Например, пусть требуется представить число 10.5 с помощью целого числа. Это невозможно, поскольку в целом числе отсутствует десятичная точка. Вы можете округлить 10.5 до 11 или обрезать его до 10, но 10.5 целым числом не является. А что, если просто умножить все числа на 10? Тогда 10.5 превратится в 105, которое является целым числом. На этом и базируются все вычисления с фиксированной точкой: вы просто умножаете числа на некоторый множитель и учитываете его в дальнейших вычислениях.

Поскольку в компьютерах используется двоичная система счисления, большинство программистов применяют 32-битовые целые числа для представления чисел с фиксированной точкой в формате 16.16, как показано на рис. 11.12.

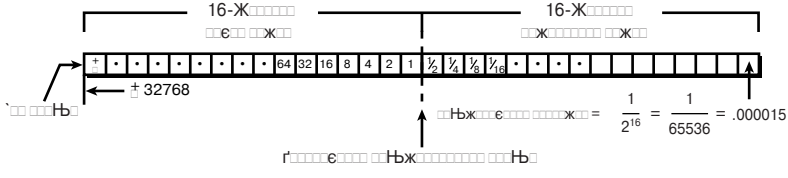


Рис. 11.12. Представление чисел с фиксированной точкой в формате 16.16

Целая часть числа размещается в старших 16 битах, а десятичная дробь — в младших 16 битах. Следовательно, вы умножаете все числа на $2^{16} = 65\,536$. Для получения целой части такого числа следует выделить старшие 16 бит, а для получения дробной части — младшие 16 бит.

Вот некоторые определения для работы с фиксированной точкой:

```
#define FP_SHIFT   16
#define FP_SCALE  65536

typedef int FIXPOINT;
```

Преобразование чисел с фиксированной точкой

Существует два типа чисел, которые требуется преобразовывать в числа с фиксированной точкой, — целые и числа с плавающей точкой. Рассмотрим каждый тип в отдельности. Для целых чисел представление является простым дополнением до 2, так что для умножения на масштабируемый множитель и преобразования в число с фиксированной точкой вы можете воспользоваться сдвигом. Числа с плавающей точкой используют формат IEEE, в котором имеются мантисса и экспонента, так что обычный сдвиг здесь неприменим и для выполнения преобразования требуется использовать умножение на масштабирующий множитель.

Дополнение до 2 является методом представления двоичных целых чисел специальным образом, обеспечивающим возможность представления как положительных, так и отрицательных чисел, а кроме того, замкнутость множества относительно арифметических операций. Дополнение до 2 двоичного числа означает инвертирование его битов и добавление 1. Допустим, вы хотите найти дополнение до 2 числа 6, т.е. получить число -6. Если для представления чисел использовать 4 бита, то 6 представляет собой двоичное число 0110, 1 — 0001, а дополнение 6 до 2 — ~0110+0001=1001+0001=1010, которое и является двоичным представлением числа -6.

Вот макросы для преобразования целых чисел и чисел с плавающей точкой в числа с фиксированной точкой:

```
#define INT_TO_FIXP(n)   (FIXPOINT((n) << FP_SHIFT))
FIXPOINT speed = INT_TO_FIXP(100);

#define FLOAT_TO_FIXP(n)  (FIXPOINT((float)n*FP_SCALE))
FIXPOINT speed = FLOAT_TO_FIXP(100.5);
```

Выделение целой и дробной частей числа с фиксированной точкой также не представляет никаких трудностей. Вот соответствующие макросы, которые извлекают старшие и младшие 16 бит из числа с фиксированной точкой:

```
#define FIXP_INT_PART(n)  (n >> FP_SHIFT)
#define FIXP_DEC_PART(n)  (n & 0x0000FFFF)
```

(Разумеется, выделенные младшие 16 бит представляют собой целое число, значение которого равно дробной части, умноженной на масштабирующий множитель; для получения реальной дробной части его следует разделить на FP_SCALE. Кроме того, не забывайте, что младшие 16 бит представляют собой беззнаковое целое число. — *Прим. ред.*).

Кроме того, вы можете воспользоваться доступом к частям посредством указателя:

```
FIXPOINT fp;

short         *integral_part = (short*)&fp+1;
unsigned short *decimal_part  = (short*)&fp;
```

(Использовать этот способ не рекомендуется в силу его непереносимости.— *Прим. ред.*).

Точность

При использовании чисел с фиксированной точкой закономерно возникает вопрос о точности. Как видно из рис. 11.12, невозможно указать дробную часть числа точнее, чем с точностью самого младшего бита, который представляет собой величину $\frac{1}{2^{16}} = \frac{1}{65536} \approx 0.000015259$. Это не так плохо для большинства приложений и обычно вполне достаточно. Хуже обстоит дело с целой частью числа. Отводящиеся для нее 16 бит означают, что диапазон представления чисел с плавающей точкой — от -32 767 до +32 768 для числа со знаком и от 0 до 65 535 для беззнакового числа. Это не очень большой диапазон, так что необходимо внимательно следить за возможностью переполнения!

Сложение и вычитание

Сложение и вычитание чисел с фиксированной точкой тривиально. Вы можете воспользоваться стандартными операторами сложения и вычитания:

```
FIXPOINT f1 = FLOAT_TO_FIX(10.5),
     f2 = FLOAT_TO_FIX(-2.6),
```

```
    f3 = 0;

// Сложение
f3 = f1 + f2;

// Вычитание
f3 = f1 - f2;
```

НА ЗАМЕТКУ Вы без проблем можете пользоваться как положительными, так и отрицательными числами в связи с тем, что для их представления использовано дополнение до 2.

Умножение и деление

Операции умножения и деления реализуются несколько сложнее, чем операции сложения и умножения. Проблема заключается в масштабированности чисел с фиксированной точкой. Соответственно, при умножении вы умножаете не только реальные числа, но и масштабирующий множитель.

$$f_1 = n_1 \cdot scale$$
$$f_2 = n_2 \cdot scale$$
$$f_3 = f_1 \cdot f_2 = (n_1 \cdot scale) \cdot (n_2 \cdot scale) = n_1 \cdot n_2 \cdot scale^2$$

Как видите, в полученном результате имеется один лишний множитель. Для его удаления нужно разделить полученное в результате число на масштабирующий множитель или, что то же, просто сдвинуть его на FP_SHIFT бит вправо:

```
f3 = ((f1*f2) >> FP_SHIFT)
```

При делении также существует проблема, но уже другого свойства.

$$f_1 = n_1 \cdot scale$$
$$f_2 = n_2 \cdot scale$$
$$f_3 = f_1 / f_2 = (n_1 \cdot scale) / (n_2 \cdot scale) = n_1 / n_2$$

Как видите, в полученном результате масштабирующий множитель отсутствует, так что полученный при делении результат следует сдвигать вправо:

```
f3 = (f1 / f2) << FP_SHIFT;
```

ВНИМАНИЕ При умножении и делении возникают проблемы переполнения и потери разрядов соответственно. При умножении результат в худшем случае может оказаться 64-битовым числом; при делении теряется точность. Возможные пути решения данной проблемы состоят в использовании 64-битовых вычислений (что при наличии процессора Pentium довольно просто осуществить на языке ассемблера) либо в изменении формата 16.16, например, на формат 24.8. Очевидно, что проблемы останутся (например, в первом случае в результате умножения можно получить 128-битовое число, а во втором резко теряется точность), но все же их острота несколько снизится.

Взгляните на имеющийся на прилагаемом компакт-диске демонстрационный пример DEMO11_4.CPP. Эта программа позволяет ввести два числа, выполняет над ними арифметические операции и показывает их результаты. Рассмотрите результаты операции деления и умножения — в данной программе используется формат 16.16 без использования 64-битовой арифметики. Затем перекомпилируйте программу для использования формата 24.8 и повторите исследования. Для перекомпиляции программы с использованием интересующего вас формата достаточно исправить макроопределения

```
#define FIXPOINT16_16
//#define FIXPOINT24_8
```

в начале программы, закомментировав первое и убрав комментарий у второго:

```
//#define FIXPOINT16_16
#define FIXPOINT24_8
```

При компиляции не забудьте, что эта программа представляет собой консольное приложение.

Развертывание цикла

Это прием оптимизации, который здорово выручал программистов во времена 8/16-битовых программ, но в настоящее время он может сыграть с вами злую шутку. Развертывание цикла означает отказ от использования цикла и выполнение всех его действий вручную. Рассмотрим, например, цикл

```
for(int index=0; index < 8; index++)
{
  sum += data[index];
}
```

Проблема заключается в том, что время выполнения действий внутри цикла оказывается меньше времени, затрачиваемого на увеличение индексной переменной, сравнение и переходы внутри цикла. Таким образом, при развертывании цикла в код

```
sum += data[0];
sum += data[1];
sum += data[2];
sum += data[3];
sum += data[4];
sum += data[5];
sum += data[6];
sum += data[7];
```

можно получить ощутимый выигрыш времени выполнения.

Однако здесь не обошлось и без подводных камней.

- Если тело цикла представляет собой сложные вычисления, время выполнения которых существенно превышает время работы механизма цикла, то не имеет смысла прибегать к развертыванию.
- При использовании процессора типа Pentium, имеющего внутренний кэш, слишком большое развертывание цикла может привести к тому, что весь его код не поместится в кэше, а следовательно, скорость вычисления цикла снизится при его развертывании. Я считаю, что развертывать следует не более 8–32 итераций цикла, в зависимости от конкретной ситуации.

Таблицы поиска

Лично мне эта оптимизация по душе больше остальных. Таблицы поиска представляют собой предварительно вычисленные значения, которые требуются нам во время работы. Вы просто вычисляете их при инициализации игры, а затем в процессе работы находите интересующие вас значения вместо их вычисления. Пусть, например, в процессе игры вам часто придется вычислять значения синуса и косинуса для углов от 0 до 359°. Частое вычисление этих функций может существенно затормозить работу игры, но если использовать таблицы поиска, вы сможете определять нужные значения за несколько

тактов процессора, так как вычисление сведется к обращению к элементу таблицы по индексу.

```
// Массивы для хранения таблиц
double SIN_LOOK[360];
double COS_LOOK[360];

// Создание таблицы поиска
for(int angle=0; angle < 360; ++angle)
{
  // Преобразуем градусы в радианы, вычисляем
  // значения sin и cos и вносим их в таблицы
  double rad_angle = angle * (3.1415926 / 180);

  SIN_LOOK[angle] = sin(rad_angle);
  COS_LOOK[angle] = cos(rad_angle);
} // for angle
```

А вот пример использования таблицы для вывода окружности радиусом 10:

```
for( int ang = 0; ang < 360; ++ang)
{
  // Вычисляем следующую точку окружности
  x_pos = 10 * COS_LOOK[ang];
  y_pos = 10 * SIN_LOOK[ang];

  // Выводим точку
  Plot_Pixel((int)x_pos+x0, (int)y_pos+y0, color);
} // for ang
```

Конечно, использование таблиц поиска приводит к дополнительным расходам памяти, но это небольшая цена за существенное повышение скорости работы. "Если можешь вычислить что-то заранее — сделай это и помести в таблицу поиска", — вот мой девиз. Как, вы думаете, работают DOOM, Quake или Half-Life?

Язык ассемблера

Последняя оптимизация, о которой пойдет речь, — использование языка ассемблера. Вы уже разработали отличный алгоритм и использовали наиболее подходящие структуры данных, но вам хочется выиграть еще несколько тактов процессора. Вы не сможете выиграть с помощью ассемблера очень много, однако в некоторых случаях выигрыш может оказаться весьма ощутимым — следует только применить ассемблер в нужном месте. Не пытайтесь использовать ассемблер, например, при написании меню — это пустая трата сил и времени. Для точного определения места, где следует применить этот мощный инструмент, воспользуйтесь профайлером. (Обычно это место находится среди кода, работающего с графикой.)

Раньше большинство компиляторов не имели встроенного ассемблера, однако в настоящее время эта возможность имеется во множестве компиляторов, включая такие распространенные компиляторы, как Microsoft, Borland или Watcom. Вот как используется встроенный ассемблер в компиляторе Microsoft Visual C++:

```
_asm
{
  // Ассемблерный код
} // _asm
```

Большим достоинством встроенного ассемблера является возможность использования имен переменных, определенных в коде C/C++. Например, вот как можно написать функцию заполнения области памяти 32-битовым значением с использованием встроенного ассемблера:

```
void qmemset(void* memory, int value, int num_quads)
{
  // В этой функции для заполнения области памяти
  // 32-битовым значением используется встроенный ассемблер
  _asm
  {
    cld             // Сброс флага направления
    mov edi, memory   // Указатель на область памяти
    mov ecx, num_quads // Счетчик цикла
    mov eax, value    // Заполняющее значение
    rep stosd         // Заполнение области памяти
  } // _asm
} // qmemset
```

Используется такая функция точно так же, как и написанная на языке программирования C/C++, например:

```
qmemset(&buffer, 25, 1000);
```

Этот вызов заполняет область памяти размером в 1000 двойных слов значением 25.

НА ЗАМЕТКУ

Если вы используете какой-то другой компилятор, обратитесь к его описанию, чтобы узнать, какой синтаксис используется в нем при работе со встроенным ассемблером. В большинстве случаев отличия заключаются в количестве используемых символов подчеркивания в ключевом слове `asm`.

Создание демоверсий

Итак, вы создали потрясающую воображение игру. Теперь вам необходима возможность *демонстрационного режима* игры. Обычно он достигается двумя способами: вы играете, записывая при этом все действия, или создаете для этого искусственный интеллект (ИИ). Разработка системы искусственного интеллекта, способного играть так же хорошо, как и человек, — задача очень сложная, так что первый способ, как более простой, распространен гораздо шире. Рассмотрим вкратце оба пути создания демоверсий.

Предварительно записанная версия

При записи демоверсии обычно производится запись и сохранение в файле состояний устройств ввода в процессе игры; затем при необходимости выполняется воспроизведение записанной информации, как будто это реальный ввод, осуществляемый игроком. Взгляните на рис. 11.13, где все это показано в виде диаграммы. Идея заключается в том, что самой игре не известно, откуда осуществляется ввод — с клавиатуры (и/или других устройств ввода) или из файла, так что воспроизведение записанной игры ничем не отличается от обычных действий игрока.

Для того чтобы такой способ работал, игра должна быть *детерминированной*. Это означает, что, сколько бы раз вы ни играли в игру, при одном и том же вводе события в ней будут развиваться совершенно одинаково. В частности, это означает, что при записи ввода следует сохранить также и значение, инициализирующее генератор случайных чисел.

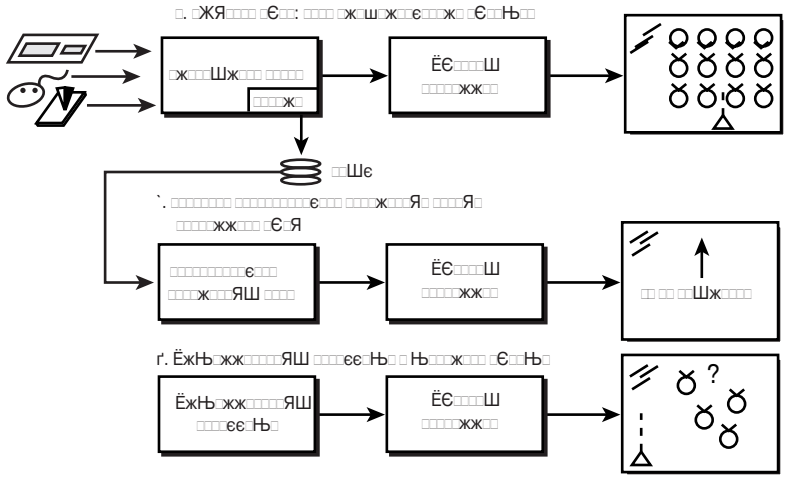


Рис. 11.13. Демоверсии игры

Наилучший подход — это запись ввода для каждого кадра игры, а не через определенные промежутки времени. В таком случае при воспроизведении записанной игры на более (или менее) быстром компьютере вам не придется дополнительно заботиться о синхронизации ввода.

Обычно я поступаю следующим образом: объединяю все устройства ввода в единую запись и сохраняю по одной такой записи для каждого кадра в файле, в начале которого я помещаю информацию о начальном состоянии игры (включая значение, инициализирующее генератор случайных чисел). Поэтому структура файла выглядит так:

```
Информация о начальном состоянии игры

Кадр 1: Входные значения
Кадр 2: Входные значения
Кадр 3: Входные значения
.
.
Кадр N: Входные значения
```

После того как файл создан, игра перезапускается, восстанавливает начальное состояние при помощи информации из файла и приступает к чтению входных данных из файла, как будто это данные, получаемые из устройств ввода. Игровой процессор просто не замечает этой подмены и не видит никакой разницы между реальной игрой и воспроизведением.

ВНИМАНИЕ

Основная ошибка новичков заключается в несвоевременном сохранении и использовании сохраненных данных, например считывании и записи в файл немного позже или немного раньше по отношению к тому моменту времени, когда выполняется чтение данных для использования в игре; таким образом, в игре и в записи могут оказаться разные данные. Для корректного воспроизведения данные, записываемые в файл, должны быть абсолютно теми же, что и использованные в записываемой игре.

Демоверсия с использованием ИИ

Второй метод создания демоверсии состоит в написании ИИ, способного играть вместо игрока. Когда игра находится в демонстрационном режиме, место игрока занимает

ИИ, подобный тем, которые используются для других персонажей игры. Одна из проблем разработки такого ИИ (не считая чисто технических сложностей) заключается в том, что он не должен обладать полной информацией об игре, т.е. знанием всех тайников, хранилищ оружия и прочих вещей. Словом, ИИ не должен знать больше, чем обычный игрок. В то же время применение ИИ отличает каждый запуск демоверсии от другого, а значит, делает его более привлекательным, чем при использовании одной предварительно записанной игры.

Реализация ИИ, подменяющего игрока, выполняется так же, как и для любого другого персонажа игры. Система ИИ подключается ко входному порту игры и подменяет обычный входной поток данных, как показано на рис. 11.13. Затем вы создаете алгоритм ИИ и определяете его основную задачу, например поиск выхода из лабиринта, уничтожение всех враждебных персонажей и т.п. После этого вы просто позволяете ИИ играть до тех пор, пока наблюдающему не захочется самому испытать себя в игре.

Стратегии сохранения игр

Одной из причин головной боли программиста может оказаться написание кода сохранения игры. Кстати, именно это разработчики игр обычно пишут в последнюю очередь.

Сохранение игры в любой момент времени означает сохранение всех ее переменных и объектов, т.е. вы должны сохранить в файле значения всех глобальных переменных и состояние каждого объекта игры. Наилучший способ решения указанной задачи — использование объектно-ориентированных технологий. Вместо разработки функции, которая записывает состояние каждого объекта, гораздо лучше обучить каждый объект записи своего состояния в файл и восстановлению этого состояния по данным, прочитанным из файла.

После этого для сохранения игры необходимо лишь записать в файл значения глобальных переменных и создать простую функцию, которая заставляет все объекты игры сохранить их состояния в этом файле. Затем, при загрузке игры из файла, вы должны загрузить значения глобальных переменных, после чего загрузить состояния всех объектов игры.

При использовании такой технологии добавление нового объекта или типа объекта позволяет вносить соответствующие изменения в сам объект, не затрагивая процесс сохранения и загрузки игры в целом.

Реализация игр для нескольких игроков

Вы хотите порадовать игроков возможностью игры “в четыре руки”, т.е. нескольких человек одновременно? Если у вас есть желание разрабатывать сетевые игры, это может стать темой отдельного большого разговора (хотя DirectPlay, как минимум, существенно облегчает вопросы коммуникаций), но сейчас речь идет о ситуации, когда вы хотите позволить двум или более игрокам одновременно играть в вашу игру или поочередно делать свои ходы (что требует только усидчивости, внимания и дополнительных структур данных).

Поочередная игра

Реализация этого подхода одновременно и проста и сложна. Она проста, поскольку если у вас есть реализация игры для одного игрока, то, по сути, чтобы реализовать ее для нескольких игроков, вам понадобится лишь внести дополнительные записи для каждого из них. Сложность же заключается в том, что вы должны обеспечить сохранение игры для каждого игрока при переключении. Очевидно, что такое сохранение должно осуществляться невидимо для игроков, но от этого его разработка не становится легче.

Вот примерный список действий в игре, где поочередно, один за другим играют два игрока.

1. Начало игры. В игру вступает игрок 1.
2. Игрок 1 играет до того момента, пока его не убивают.
3. Сохраняется состояние игры для игрока 1. В игру вступает игрок 2.
4. Игрок 2 играет до того момента, пока его не убивают.
5. Сохраняется состояние игры для игрока 2.
6. Загружается предварительно сохраненная игра игрока 1, который вновь вступает в игру.
7. Переходим к шагу 2.

Понятно, что, если вы хотите реализовать игру большего числа игроков, просто позвольте им играть по очереди до конца списка, а затем начните список сначала.

Разделение экрана

Обеспечить возможность одновременной игры на одном экране для двух и более игроков сложнее, чем в случае их поочередной игры, поскольку при этом потребуется, как минимум, рассмотреть вопросы взаимодействия игроков на всех уровнях — от сценария игры до кодирования. Кроме того, каждого игрока следует обеспечить своим устройством ввода (например, джойстиком или одного джойстиком, а другого — клавиатурой; возможно также использование разных клавиш одной клавиатуры).

Другой проблемой оказывается неприспособленность ряда игр для многопользовательского режима. Например, один игрок может пойти по одному пути лабиринта, а другой — по другому. Такой проблемы обычно не возникает в играх, где игроки перемещаются группой или всегда располагаются относительно близко друг к другу.

Если же вы хотите предоставить игрокам большую свободу действий, то для этого можно воспользоваться технологией разделения экрана, показанной на рис.11.14.

Рис. 11.14. Разделение экрана игры

Главной сложностью при разделении экрана является... разделение экрана. Вы должны генерировать два или большее количество изображений игры, что само по себе доста-

точно сложно технически. Кроме того, размер экрана может оказаться слишком мал, чтобы разместить на нем несколько виртуальных экранов для каждого из игроков, и играть станет очень сложно. И тем не менее, если вы сможете реализовать многопользовательский режим игры, она от этого только выиграет.

Многопоточное программирование

До этого момента все демонстрационные программы в данной книге использовали однопоточный цикл событий и соответствующую модель программирования. Цикл событий реагировал на ввод игрока и осуществлял вывод на экран со скоростью порядка 30 кадров в секунду. Наряду с реакцией на пользовательский ввод в игре выполняются миллионы операций для решения десятков, если не сотен, небольших задач, таких, как вывод объектов, обработка вводимой информации, создание звукового сопровождения и многое другое. На рис. 11.15 показан используемый нами стандартный цикл игры.

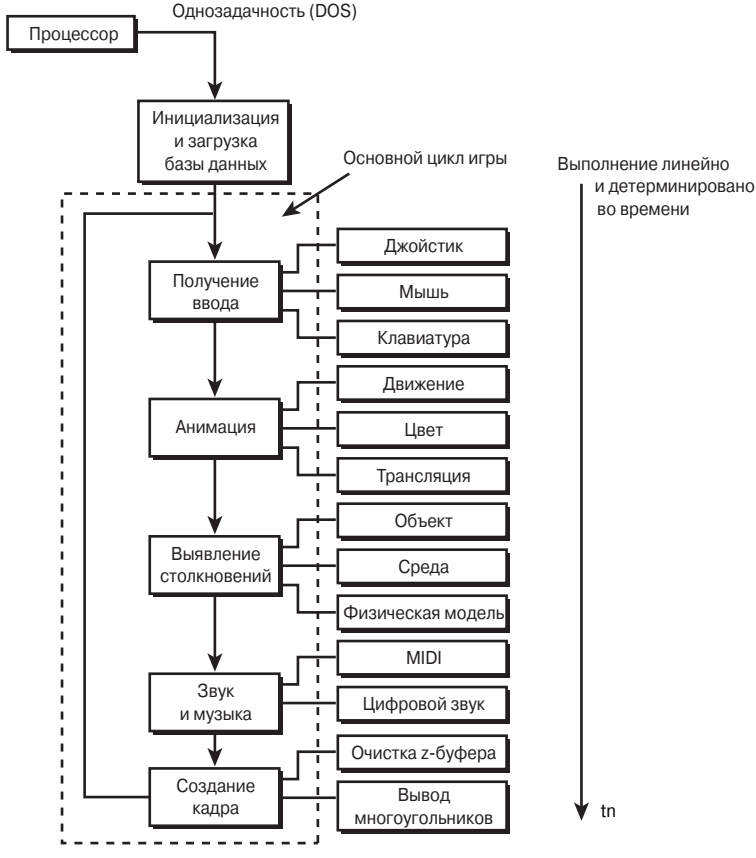


Рис. 11.15. Стандартный однозадачный цикл игры

Как видно из рис. 11.15, вся логика игры выполняется последовательно. Конечно, из этого правила имеются исключения, например прерывания, связанные с выводом музыки или обслуживанием устройств ввода, но в основном игра представляет собой одну бесконечно повторяемую длинную последовательность вызовов функций.

Однако внешне игра выглядит достаточно живо и реалистично, несмотря на последовательный характер выполнения. Дело в том, что высокая скорость работы компьютера обеспечивает кажущуюся одновременность выполнения разных задач. Модель, используемая большинством программистов, представляет собой не что иное, как однозадачный поток, который последовательно выполняет различные действия для получения необходимого вывода на экран в каждом кадре. Это одна из простейших моделей, использующаяся в программировании в DOS.

Однако времена DOS отошли в прошлое, и современное программирование все чаще использует многопоточные возможности Windows 95/98/ME/XP/NT/2000.

В этом разделе речь идет о *потоках выполнения* в операционных системах Windows 95/98/NT, которые позволяют одновременно выполнять несколько заданий в рамках одного и того же приложения. Для начала рассмотрим используемую в этой области терминологию.

Терминология многопоточного программирования

В компьютерном лексиконе имеется, пожалуй, слишком много слов, начинающихся с "много-". Познакомимся с некоторыми из них, в частности выясним, что такое многопроцессорность, многозадачность и многопоточность[3].

Многопроцессорный компьютер — это компьютер с более чем одним процессором. Примерами таких компьютеров могут служить Cray или Connection Machine, который может иметь до 64000 процессоров.

Однако опустимся с небес на землю, где вполне можно позволить себе купить компьютер с четырьмя процессорами Pentium III+ и установить на нем операционную систему Windows NT. Это будет типичный представитель симметричной многопроцессорной системы (SMP), т.е. системы, где все четыре процессора будут работать симметрично. Это не совсем верно, поскольку ядро операционной системы будет работать только на одном из процессоров, однако при запуске нового процесса он может с одинаковым успехом выполняться любым из процессоров. Таким образом, основная идея использования многопроцессорных компьютеров состоит в разделении загрузки между процессорами.

В некоторых системах на каждом процессоре может выполняться только одна задача или процесс; в других системах, таких, как Windows NT, на каждом процессоре могут выполняться тысячи задач. В этом и заключается суть *многозадачности*, представляющей собой одновременное выполнение множества задач одно- или многопроцессорной машиной.

Теперь рассмотрим концепцию *многопоточности*, которая в настоящий момент интересует нас больше всего. Процесс в операционной системе Windows 95/98/NT/2000 представляет собой программу, которая может иметь собственное адресное пространство и контекст и существовать сама по себе.

Поток (thread) представляет собой более простую единицу выполнения. Потоки создаются процессами и работают в адресном пространстве создавшего их процесса. Это приводит к тому, что организация связи потоков в пределах одного процесса осуществляется очень просто. Поток, который может выполнять какие-то действия параллельно основной задаче, при этом не требует постоянного присмотра и к тому же имеет доступ к глобальным переменным в вашей программе, — это именно то, что нужно вам, как программисту, работающему над играми.

[3] Для получения более детальной информации о процессах, потоках и устройстве операционных систем обратитесь к книге Столлингс В. *Операционные системы, 4-е издание*. — М. : Издательский дом "Вильямс", 2002. — *Прим. ред.*

Наряду с концепциями, описываемыми терминами, начинающимися на "много-", следует иметь представление и о некоторых других вещах. Операционные системы Windows 95/98/NT/2000 представляют собой многозадачные вытесняющие операционной системы. Это означает, что ни одна задача, процесс или поток не могут получить компьютер в свое распоряжение целиком — каждый из них будет в некоторый момент времени вытеснен и заблокирован и начнется выполнение очередного потока. Работа этих операционных систем в корне отличается от работы Windows 3.1, которая *не* являлась вытесняющей. Если вы не выполняли в каждой итерации цикла вызов `GetMessage(...)`, другие процессы в системе не могли выполняться. В Windows 95/98/NT/2000 вы можете запустить бесконечный цикл, не выполняющий никаких действий, но от этого выполнение других задач не прекратится.

Кроме того, в операционных системах Windows 95/98/NT/2000 каждый процесс и поток обладает приоритетом, который указывает, как долго этот процесс или поток будет выполняться до вытеснения. Так, если имеется 10 потоков с одинаковым приоритетом, то все они по очереди получат одинаковое количество процессорного времени. Однако если один поток имеет приоритет уровня ядра, то в каждом цикле он будет выполняться чаще остальных (рис. 11.16).

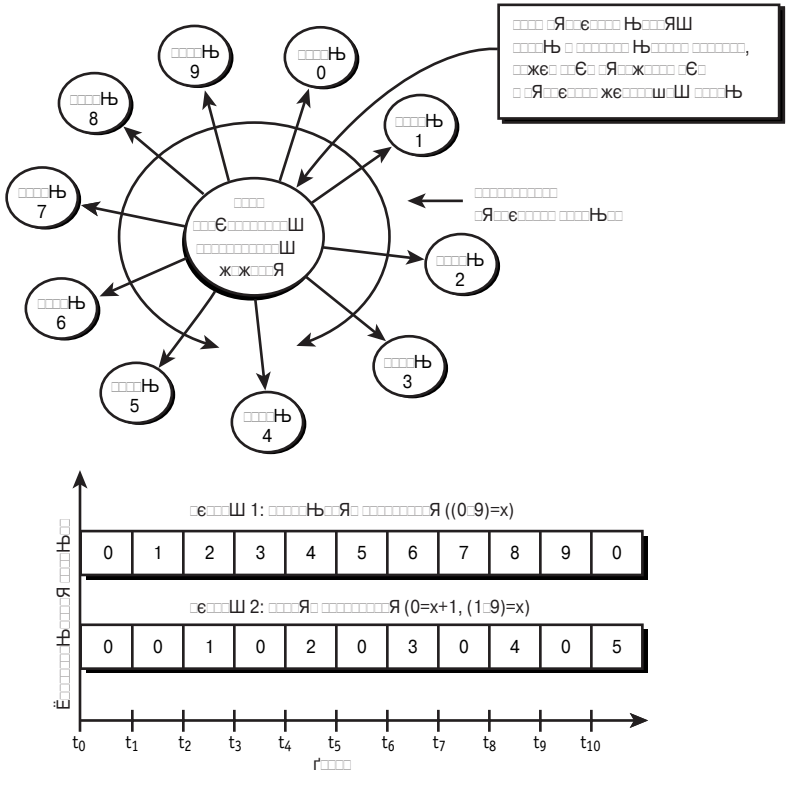


Рис. 11.16. Поочередное выполнение потоков с равными и неравными приоритетами

И наконец, рассмотрим, в чем же состоит отличие между многопоточностью операционных систем Windows 95/98/NT/2000. Между ними действительно имеются определенные различия, но они столь невелики, что модель потоков Windows 95 вполне может использоваться на всех платформах Windows. Хотя многопоточность в Windows 98 и NT

надежнее и интеллектуальнее, в большинстве примеров этого раздела я буду использовать многопоточность Windows 95.

Зачем нужны потоки в играх?

Ответ на этот вопрос должен быть для вас очевиден. Я думаю, вы согласитесь, что существуют сотни вещей, которые гораздо проще осуществить с применением потоков, чем без них. Вот наиболее распространенные задачи, для решения которых стоит использовать потоки:

- обновление кадров анимации;
- создание звуковых эффектов;
- управление малыми объектами;
- получение информации от устройств ввода;
- обновление глобальных структур данных;
- создание всплывающих меню и управляющих элементов.

Последнее применение — одно из моих любимых. Я всегда стараюсь создавать всплывающие меню, при работе с которыми игра продолжается, а добиться этого с использованием потоков гораздо проще.

Прежде чем приступить к созданию вашего первого потока, четко уясните следующее: в однопроцессорном компьютере в каждый момент времени выполняется только один поток. Говорить об одновременности выполнения в этом случае можно только в том смысле, что частое чередование выполнения потоков создает иллюзию одновременности работы нескольких потоков. Добиться ускорения работы приложения в целом путем использования потоков невозможно, но их применение делает программирование сложных задач более простым и корректным. На рис. 11.17 показан пример основного процесса, который порождает три потока, выполняющихся вместе с ним.

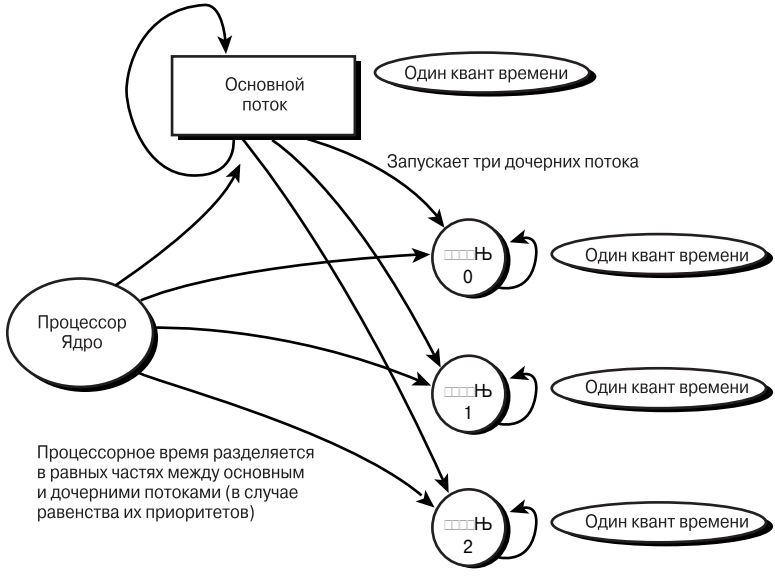


Рис. 11.17. Процесс, создающий три дочерних потока

Однако хватит слов, пора приступать к делу.

Создание потоков

Сейчас в качестве демонстрационных программ мы будем использовать консольные приложения, так что не забудьте установить корректные опции вашего компилятора. (Я, наверное, уже надоел вам этими напоминаниями, но поверьте — сотни возмущенных писем о том, что мои программы не компилируются и не работают (только потому, что авторы этих писем забыли поставить соответствующие опции компилятора!), надоедают мне гораздо больше.)

Кроме того, при создании этих демонстрационных приложений следует использовать *многопоточные* библиотеки. Это можно сделать с помощью главного меню MS DEV Studio (Project⇨Settings), затем во вкладке C/C++ перейти к категории Code Generation и выбрать в списке Use run-time library пункт Multithreaded. Все это показано на рис. 11.18. Кроме того, отключите оптимизацию — иногда она мешает корректной синхронизации.

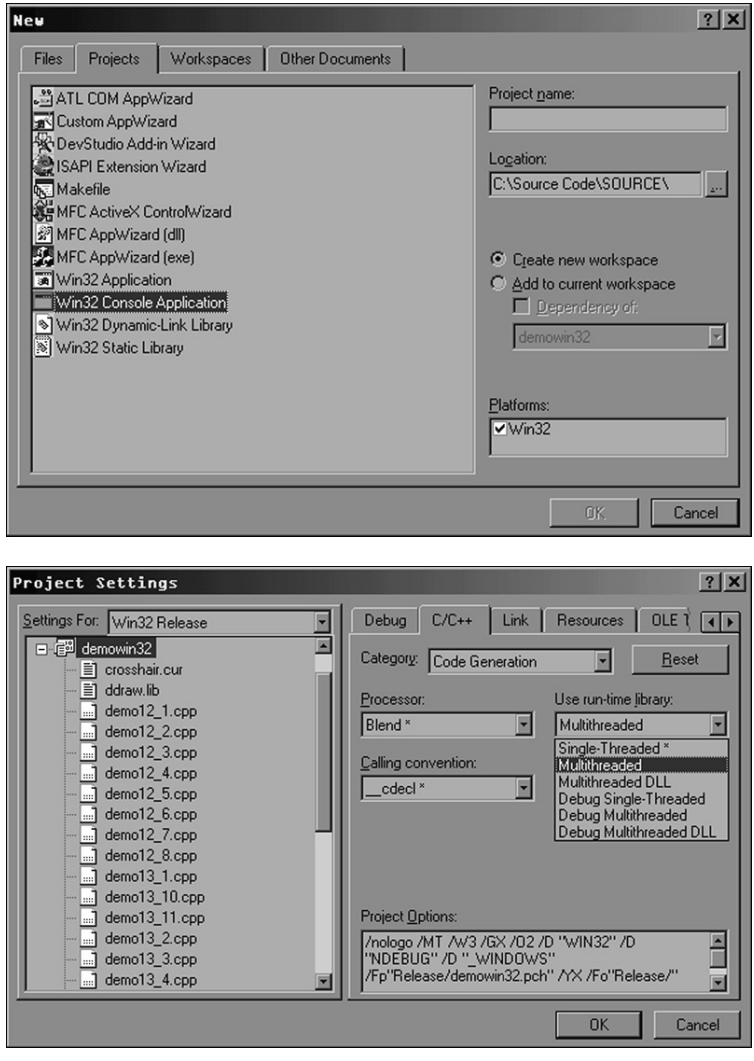


Рис. 11.18. Создание консольного приложения с многопоточными библиотеками

Теперь, когда, надеюсь, все установлено совершенно правильно, можно начать работу с кодом. Итак, создание потока — дело совсем несложное. Для этого используется следующий вызов Win32 API:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
    // Указатель на атрибуты безопасности
  DWORD              dwStackSize,
    // Начальный размер стека потока (байт)
  LPTHREAD_START_ROUTINE lpStartAddress,
    // Указатель на функцию потока
  LPVOID             lpParameter,
    // Аргумент нового потока
  DWORD              dwCreationFlags,
    // Флаги создания
  LPDWORD            lpThreadId);
    // Указатель на возвращаемый идентификатор потока
```

Параметр lpThreadAttributes указывает на структуру LPSECURITY_ATTRIBUTES, в которой определены атрибуты безопасности для создаваемого потока. Если значение lpThreadAttributes равно NULL, поток создается с дескриптором безопасности по умолчанию.

Параметр dwStackSize определяет размер стека нового потока в байтах. Если он имеет нулевое значение, используется то же значение размера стека, что и для главного потока процесса. Стек автоматически выделяется в области памяти процесса и освобождается при завершении потока. Заметим, что при необходимости размер стека может увеличиваться. Функция CreateThread пытается выделить необходимый блок памяти для стека и, если это не удается, завершается неуспешно.

lpStartAddress указывает на функцию, которая выполняется потоком и представляет его стартовый адрес. Эта функция должна получать один 32-битовый аргумент и возвращать 32-битовое значение.

lpParameter представляет собой 32-битовое значение, передаваемое функции потока в качестве параметра.

dwCreationFlags указывает дополнительные флаги, управляющие созданием потока. Если определен флаг CREATE_SUSPENDED, поток создается в приостановленном состоянии и не будет выполняться до тех пор, пока не будет осуществлен вызов функции ResumeThread(). Если это значение равно 0, поток начинает выполняться непосредственно после создания.

lpThreadId указывает на 32-битовую переменную, в которой размещается идентификатор созданного потока.

При успешном завершении функция CreateThread() возвращает значение дескриптора нового потока. При неуспешном завершении функции возвращается значение NULL. Для получения расширенной информации об ошибке воспользуйтесь функцией GetLastError().

Хотя функция CreateThread() и выглядит несколько сложно, на самом деле бо́льшая часть ее функциональности обычно не требуется.

По завершении работы потока необходимо закрыть его дескриптор; другими словами, нужно дать знать системе, что вы завершили использование объекта. Для этого следует вызвать функцию CloseHandle(), которой необходимо передать дескриптор потока, возвращенный функцией CreateThread().

Эти действия следует выполнять для каждого потока, работа которого завершена. Однако этот вызов не уничтожает поток — он просто сообщает системе, что поток больше не существует. Работа потока может завершиться при выходе из функции потока вызо-

вом функции TerminateThread() либо операционной системой при завершении работы главного потока приложения. Обо всем этом речь пойдет позже, а пока достаточно знать, что освобождение захваченных при создании потока ресурсов осуществляется вызовом функции CloseHandle() со следующим прототипом:

```
BOOL CloseHandle(HANDLE hObject);
```

Параметр hObject представляет собой дескриптор объекта. Если функция завершается успешно, она возвращает значение TRUE, при неуспешном завершении возвращается значение FALSE. Для получения в последнем случае расширенной информации об ошибке воспользуйтесь функцией GetLastError(). Кроме потоков, функция CloseHandle() используется для закрытия дескрипторов следующих объектов:

- консольного ввода или вывода;
- отображаемых файлов;
- взаимоисключений;
- именованных каналов;
- процессов;
- семафоров.

Функция CloseHandle() делает недействительным переданный ей дескриптор объекта и уменьшает значение счетчика дескрипторов объекта. Когда закрывается последний дескриптор, объект удаляется из операционной системы.

ВНИМАНИЕ

По умолчанию дескриптор нового потока разрешает полный доступ к созданному потоку. Если дескриптор безопасности при создании потока отсутствует, дескриптор потока может использоваться любой функцией, которой требуется дескриптор потока. Если же при создании потока был указан дескриптор безопасности, то при любой попытке использования дескриптора потока выполняется проверка прав доступа к нему.

Далее рассмотрим, как может выглядеть код, представляющий поток, создаваемый функцией CreateThread().

```
DWORD WINAPI My_Thread(LPVOID data)
{
  // ... выполнение некоторых действий ...

  // возврат некоторого значения
  return(25);
}
```

Теперь у вас есть все необходимое для создания первого многопоточного приложения. Первый пример иллюстрирует создание одного дополнительного потока наряду с основным. Основной поток выводит на экран число 2, а дополнительный — число 1. Полностью код данного приложения содержится в файле DEMO11_5.CPP на прилагаемом компакт-диске.

```
// DEMO11_5.CPP - Создание потока для одновременного вывода
// символов двумя потоками.

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

#define WIN32_LEAN_AND_MEAN
```

```
#include <windows.h>
#include <windowsx.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

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

DWORD WINAPI Printer_Thread(LPVOID data);

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

DWORD WINAPI Printer_Thread(LPVOID data)
{
   // Этот поток состоит в выводе переданной информации
   // 25 раз с небольшой задержкой

   for (int index=0; index<25; index++)
   {
      printf("%d ",data); // Вывод символа
      Sleep(100);            // Задержка
   } // for index


   // Возвращаем переданные в функцию данные

   return((DWORD)data);

} // Printer_Thread

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

void main(void)
{

   HANDLE thread_handle;  // Дескриптор потока
   DWORD  thread_id;         // Идентификатор потока


   // Начинаем с пустой строки

   printf("\nStarting threads...\n");

   // Создание потока

   thread_handle = CreateThread(
      NULL,              // безопасность по умолчанию
      0,                 // стек по умолчанию
      Printer_Thread, // функция потока
```

```
    (LPVOID)1,    // пользовательские данные,
                  // передаваемые потоку
    0,            // флаг создания
    &thread_id); // идентификатор потока

  // Вход в цикл вывода — более длинный для гарантии
  // того, что создаваемый поток завершится первым

  for (int index=0; index<50; index++)
  {
    printf("2 ");
    Sleep(100);
  } // for index

  // к этому моменту поток уже завершает работу
  CloseHandle(thread_handle);

  printf("\nAll threads terminated.\n");

} // main
```

Вот как выглядит возможный вывод данной программы:

```
Starting threads...
2 1 2 1 2 1 2 2 1 1 2 1 2 1 2 1 2 2 2 1 2 1 2 1 2 1
2 2 1 1 2 1 2 1 2 1 2 2 1 2 1 2 1 2 1 2 2 1 1 2 1
2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
All threads terminated.
```

Как видно из приведенного вывода программы, каждый из потоков выполняется в течение некоторого короткого времени, после чего операционная система переключается на выполнение следующего ожидающего потока. В нашем случае таких потоков всего два, и операционная система переключается между выполнением основного и дополнительного потоков.

Теперь попытаемся создать сразу несколько потоков. Для этого нужно лишь немного изменить функциональность программы DEMO11_5.CPP. Все, что требуется, — это обеспечить неоднократный вызов функции CreateThread(). Кроме того, каждому потоку передаются свои данные, с тем чтобы потоки можно было различить по их выводу на экран. Измененная соответствующим образом программа находится на прилагаемом компакт-диске в файле DEMO11_6.CPP. Обратите внимание на использование массивов для хранения дескрипторов и идентификаторов потоков.

```
// DEMO11_6.CPP - Создание трех дочерних потоков

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

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
```

```
#include <math.h>
#include <io.h>
#include <fcntl.h>

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

#define MAX_THREADS 3

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

DWORD WINAPI Printer_Thread(LPVOID data);

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


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

DWORD WINAPI Printer_Thread(LPVOID data)
{
  // Этот поток состоит в выводе переданной информации
  // 25 раз с небольшой задержкой

  for (int index=0; index<25; index++)
  {
    printf("%d ",data); // Вывод символа
    Sleep(100);         // Задержка
  } // for index


  // Возвращаем переданные в функцию данные

  return((DWORD)data);

} // Printer_Thread

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

void main(void)
{

  HANDLE thread_handle[MAX_THREADS];
    // Массив для хранения дескрипторов потоков
  DWORD  thread_id[MAX_THREADS];
    // Массив для хранения идентификаторов потоков

  // Начинаем с пустой строки
  printf("\nStarting all threads...\n");

  // Создаем потоки

  for (int index=0; index<MAX_THREADS; index++)
  {
    thread_handle[index] = CreateThread(
```

```
        NULL,                  // безопасность по умолчанию
        0,                     // стек по умолчанию
        Printer_Thread,        // функция потока
        (LPVOID)index,         // пользовательские данные,
                               // передаваемые потоку
        0,                     // флаг создания
        &thread_id[index]); // идентификатор потока

    } // for index

    // Вход в цикл вывода — более длинный для гарантии
    // того, что создаваемый поток завершится первым

    for (index=0; index<75; index++)
    {
       printf("4 ");
       Sleep(100);

    } // for index

    // Все потоки к этому моменту должны завершиться

    for (index=0; index<MAX_THREADS; index++)
       CloseHandle(thread_handle[index]);

    printf("\nAll threads terminated.\n");

} // main
```

Вот как выглядит возможный вывод данной программы:

```
Starting all threads...
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 4 1 2 3 4 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 1 2 3 4 1 2 3 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
All threads terminated.
```

Вот видите, как легко и просто создать несколько потоков! Обратите внимание на то, что, хотя для всех создаваемых потоков использовался один и тот же код, вывод каждого из них отличается от вывода других потоков. Дело в том, что все переменные в коде потока размещаются в стеке, а каждый поток имеет собственный стек. Схематично это изображено на рис. 11.19.

На рис. 11.19 отсутствует такая важная вещь, как завершение потоков. Потоки завершаются самостоятельно, и основной поток никак не управляет этим процессом. Кроме того, основной поток не имеет никаких подтверждений тому, что дочерний поток завершил свою работу (соответственно, не может и получить возвращенное этим потоком значение).

Для исправления этой ситуации необходим способ обмена информацией между потоками и проверки состояния одного потока из другого. Для завершения работы потока имеется такой грубый метод, как использование функции TerminateThread(), но я бы не рекомендовал его применять.

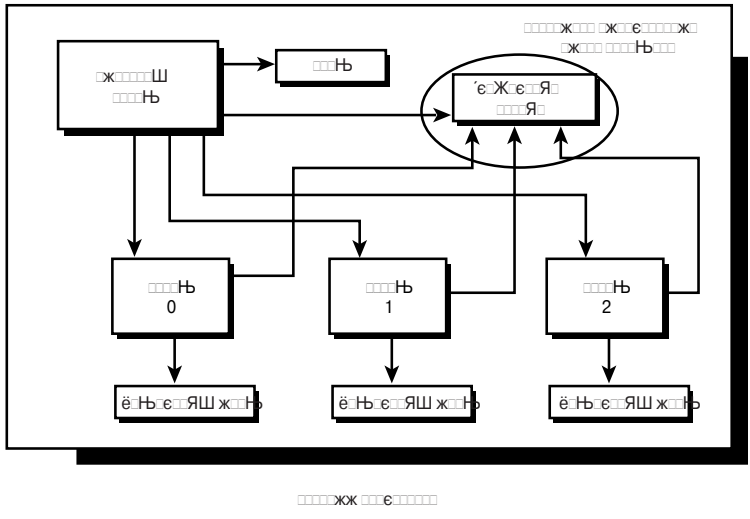


Рис. 11.19. Распределение памяти и кода основного и вторичного потоков

Пересылка сообщений между потоками

Что делать, если требуется обеспечить возможность управления дочерними потоками со стороны основного потока? Например, необходимо, чтобы основной поток мог прекратить работу всех дочерних потоков. Как это можно сделать?

Этого можно добиться разными способами.

- Посылкой потоку сообщения о том, что он должен завершить свою работу (корректный способ).
- Вызовом функции уровня ядра, который приведет к завершению работы потока (некорректный способ).

Хотя последний, некорректный, способ и необходим в ряде случаев, пользоваться им небезопасно в силу того, что он просто “убивает поток из-за угла”, не давая тому выполнить необходимые процедуры для освобождения захваченных ресурсов. При этом могут возникнуть такие неприятности, как утечка памяти или других ресурсов, так что будьте осторожны!

На рис. 11.20 показаны оба упомянутых метода завершения потоков.

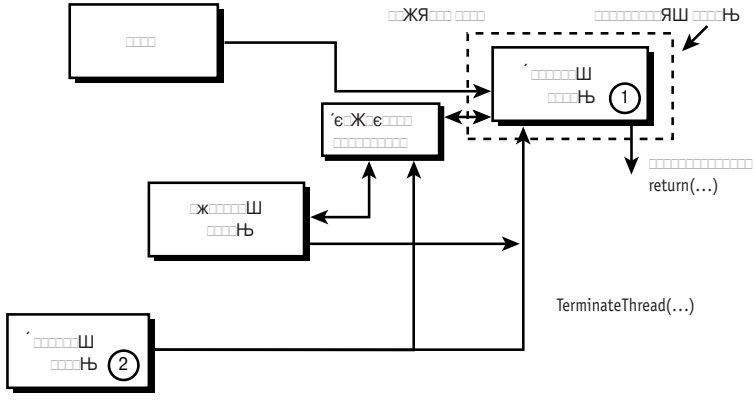


Рис. 11.20. Методы завершения потоков

Прежде чем перейти к рассмотрению пересылки сообщений между потоками для завершения их работы, рассмотрим функцию TerminateThread(), чтобы вы знали, как использовать ее в случае крайней необходимости.

```
BOOL TerminateThread(HANDLE hThread,  // Дескриптор потока
          DWORD  dwExitCode);         // Код выхода потока
```

Параметр hThread указывает, какой именно поток следует завершить. Для его завершения необходимо иметь права доступа THREAD_TERMINATE.

Параметр dwExitCode определят код выхода потока, который может быть получен при помощи вызова GetExitCodeThread().

В случае успешного завершения функция TerminateThread() возвращает значение TRUE, в противном случае — FALSE. Для получения расширенной информации о происшедшей ошибке воспользуйтесь функцией GetLastError().

Функция TerminateThread() используется для немедленного завершения работы. После ее вызова у потока нет никакого шанса выполнить хотя бы несколько инструкций. Его стек не освобождается, а DLL, присоединенные к потоку, не получают уведомления о прекращении работы потока. Как вы понимаете, все это — очень нездоровое поведение.

Поэтому перейдем к более корректному методу завершения работы потоков — пересылке им соответствующего уведомляющего сообщения. Конечно, "пересылка" звучит громко, на самом деле это всего лишь установление некоторого предопределенного значения глобальной переменной-флага, состояние которой опрашивается дочерним потоком. Когда этот поток обнаруживает, что пора "закругляться", он выполняет все необходимые действия по освобождению захваченных ресурсов и корректному завершению работы. Но как основной поток может узнать о завершении работы всех дочерних потоков? Для этого можно использовать другую глобальную переменную, значение которой уменьшается завершающимися потоками, т.е., по сути, счетчик количества работающих потоков.

Значение этого счетчика может быть проверено основным потоком: если оно равно 0, это означает, что все дочерние потоки завершили свою работу, а следовательно, основной поток может закрывать их дескрипторы. Это *почти* корректный способ. Почему "почти"? Поговорим об этом после того, как рассмотрим код демонстрационной программы DEMO11_7.CPP.

```
// DEMO11_7.CPP - Использование глобальной переменной для
//          управления завершением работы потоков.

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

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

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

#define MAX_THREADS 3
```

```
//////////////////////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

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

int terminate_threads = 0;   // Глобальный флаг завершения
int active_threads    = 0;   // Количество активных потоков[4]

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

DWORD WINAPI Printer_Thread(LPVOID data)
{
  // Эта функция просто выводит переданную ей информацию
  // до тех пор, пока не получит сигнал о необходимости
  // завершения работы

  for(;;)
  {
    printf("%d ",(int)data+1); // Вывод символа
    Sleep(100);

    // Проверка флага завершения
    if (terminate_threads)
      break;

  } // for index

  // Уменьшение количества активных потоков
  if (active_threads > 0)
    active_threads--;

  // Возврат переданного функции значения
  return((DWORD)data);

} // Printer_Thread

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

void main(void)
{

  HANDLE thread_handle[MAX_THREADS]; // Дескрипторы потоков
  DWORD  thread_id[MAX_THREADS];     // Идентификаторы потоков

  printf("\nStarting Threads...\n");

  // Создание потоков
  for (int index=0; index < MAX_THREADS; index++)
  {
    thread_handle[index] = CreateThread(
      NULL,                 // безопасность по умолчанию
      0,                    // стек по умолчанию
      Printer_Thread,       // функция потока
      (LPVOID)index,        // пользовательские данные,
```

[4] Данную переменную лучше описать как volatile, поскольку иначе оптимизирующий компилятор может некорректно оптимизировать цикл проверки ее значения. — *Прим. ред.*

```
                                // передаваемые потоку
        0,                      // флаг создания
        &thread_id[index]);  // идентификатор потока

      // Увеличиваем счетчик потоков
      active_threads++;

   } // for index

   // Входим в цикл вывода основного потока

   for (index=0; index<25; index++)
   {
      printf("4 ");
      Sleep(100);
   } // for index

   // В этот момент все потоки продолжают выполнение.
   // Теперь при нажатии клавиши на клавиатуре всем
   // потокам будет послано уведомление о необходимости
   // завершения работы, и основной поток должен просто
   // дождаться завершения выполнения всех дочерних потоков.

   while(!kbhit());

   // Получение значения введенного символа
   getch();

   // Установка флага завершения
   terminate_threads = 1;

   // Ожидаем завершения всех потоков (active_threads==0)
   while(active_threads);

   // К этому моменту все потоки завершили работу, так что
   // мы закрываем их дескрипторы
   for (index=0; index < MAX_THREADS; index++)
      CloseHandle(thread_handle[index]);

   printf("\nAll threads terminated.\n");

} // main
```

Вот как выглядит возможный вывод данной программы:

```
Starting Threads...
4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 4 1 2 3 4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
1 2 3 4 1 2 3 4 4 1 2 3 4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 4 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 1 2 1 2 3 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 3 1 3 1
All threads terminated.
```

Как видно из приведенного вывода программы, когда пользователь нажимает клавишу, все потоки завершают работу, после чего прекращается выполнение основного потока программы. Однако при использовании этого метода есть два проблемных момента.

Первая проблема не вполне очевидна. Приведем пример сценария, который выявляет данную проблему.

1. Все дочерние потоки, кроме одного, завершили работу.
2. Последний поток уменьшил значение глобального счетчика потоков, которое стало равно 0.
3. В этот момент произошло переключение потоков. Продолжающий выполнение основной поток обнаруживает, что значение счетчика равно 0, и закрывает все дескрипторы, включая дескриптор потока, который все еще не завершил свою работу!

(Возможен еще более неприятный сценарий.

1. Поток 1 считывает значение счетчика (например, равное 2) в память. Происходит переключение потоков.
2. Поток 2 считывает значение счетчика (все еще равное 2) в память. Происходит переключение потоков.
3. Поток 1 уменьшил значение счетчика в памяти до 1 и записал его в глобальную переменную.
4. Поток 2 уменьшил значение счетчика в памяти до 1 и записал его в глобальную переменную.

В результате, несмотря на то что свою работу завершили два потока, значение счетчика уменьшилось на 1, так что в этой ситуации оно никогда не станет равным 0, а следовательно, программа не завершит свою работу. — *Прим. ред.*)

Таким образом, нам необходима специальная функция, выясняющая, завершена ли работа потока. Win32 API предоставляет ряд функций Wait*(), которые могут в этом помочь.

Вторая проблема состоит в том, что в основном потоке имеется цикл ожидания, пока счетчик потока не примет нулевое значение. Использование такого цикла вполне допустимо в Win16/DOS, но не в Win32. Здесь это просто пустая трата процессорного времени (хотя бы по той очевидной причине, что в течение кванта времени, отпущенного основному потоку, другие потоки работать не могут, так что значение счетчика в течение этого кванта времени измениться не может. Следовательно, циклическая проверка — пустая трата кванта времени). Чтобы убедиться в этом, можно воспользоваться системной утилитой типа SYSMON.EXE (Windows 95/98/ME/XP) или PERFMON.EXE (Windows NT/2000) либо подобными им, показывающими загрузку процессора. Описанная проблема также решается с помощью функций Wait*().

Ожидание завершения потоков

Когда любой поток завершает свою работу, он *сигнализирует* об этом ядру. Неважно, что именно обозначает этот термин — важно, как узнать об этом событии.

Вы можете проверить наличие сигнала с помощью функций Wait*(), которые позволяют проверить наличие одного или нескольких сигналов. Кроме того, вы можете использовать функции Wait*() для ожидания сигнала, однако без транжирящего процессорное время цикла проверки. Такой способ в большинстве случаев гораздо лучше непрерывной проверки значения глобальной переменной. На рис. 11.21 показан механизм работы функций Wait*() и их взаимоотношения с работающим приложением и ядром операционной системы.

Наконец настало время узнать, что же скрывается за этим Wait*(). Это функции WaitForSingleObject() и WaitForMultipleObjects(), которые используются для ожидания одного или нескольких сигналов соответственно. Вот определения этих функций:

```
DWORD WaitForSingleObject(
    HANDLE hHandle,          // Дескриптор объекта
    DWORD  dwMilliseconds); // Интервал ожидания (ms)
```

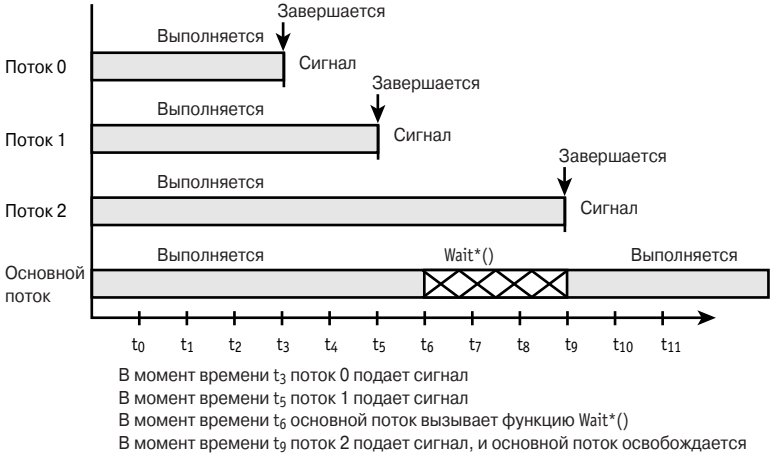


Рис. 11.21. Временная диаграмма работы функций Wait()*

Параметр hHandle указывает объект, сигнал от которого мы ожидаем, а dwMilliseconds — интервал времени ожидания в миллисекундах. Функция завершает работу по истечении этого интервала, даже если объект так и не подал сигнал. Значение INFINITE соответствует бесконечному интервалу времени ожидания.

При успешном завершении функции возвращаемое значение указывает, какое событие заставило функцию завершить работу; при неуспешном завершении возвращается значение WAIT_FAILED, а дополнительную информацию о происшедшей ошибке можно получить при помощи вызова GetLastError().

В случае успешного завершения возвращается одно из приведенных ниже значений.

- WAIT_ABANDONED. Объект представляет собой взаимоисключение, или мьютекс, который не был освобожден владеющим им потоком, а сам поток завершил свою работу. Владение мьютексом передается вызывающему потоку, и мьютекс устанавливается в несигнализирующее состояние.
- WAIT_OBJECT_0. Интересующий нас объект находится в сигнализирующем состоянии.
- WAIT_TIMEOUT. Переданный функции интервал времени истек, а объект при этом остается в несигнализирующем состоянии.

Функция WaitForSingleObject() проверяет текущее состояние указанного объекта. Если объект находится в несигнализирующем состоянии, вызывающий поток входит в состояние ожидания, однако это ожидание гораздо более эффективно, чем циклический опрос значения переменной. Поток при этом использует очень небольшое процессорное время.

Ожидание нескольких сигналов реализуется с помощью функции WaitForMultipleObjects().

```
DWORD WaitForMultipleObjects(
  DWORD      nCount,          // Количество дескрипторов
                              // в массиве
  CONST HANDLE *lpHandles,    // Массив дескрипторов
  BOOL       bWaitAll,        // Флаг ожидания
  DWORD      dwMilliseconds); // Интервал ожидания (ms)
```

Параметр nCount — это количество дескрипторов объектов в массиве, на который указывает параметр lpHandles. Максимальное число дескрипторов объектов в массиве — MAXIMUM_WAIT_OBJECTS.

lpHandles указывает на массив дескрипторов объектов, который может содержать дескрипторы объектов разных типов. (Примечание: в случае использования Windows NT дескрипторы должны иметь права доступа SYNCHRONIZE.)

bWaitAll определяет тип ожидания. Если значение этого параметра равно TRUE, возврат из функции происходит тогда, когда сигналы подадут все объекты из массива lpHandles. Если же значение параметра равно FALSE, возврат из функции произойдет при подаче сигнала любым из объектов. В этом случае возвращаемое функцией значение указывает, сигнал от какого объекта привел к завершению функции.

dwMilliseconds указывает интервал времени ожидания в миллисекундах. Функция завершает работу по истечении этого интервала, даже если условие, определяемое параметром bWaitAll, так и не наступило. Значение INFINITE соответствует бесконечному интервалу времени ожидания.

В случае успешного завершения функции возвращаемое значение указывает, какое именно событие вызвало его. Если же функция завершается неуспешно, возвращается значение WAIT_FAILED, а дополнительную информацию о происшедшей ошибке можно получить при помощи вызова GetLastError().

В случае успешного завершения возвращается одно из приведенных ниже значений.

- От WAIT_OBJECT_0 до (WAIT_OBJECT_0+nCount-1). Если bWaitAll равно TRUE, это возвращаемое значение указывает, что все объекты находятся в сигнализирующем состоянии. Когда bWaitAll равно FALSE, возвращаемое значение, уменьшенное на величину WAIT_OBJECT_0, представляет собой индекс в массиве lpHandles того объекта, который подал сигнал. Если в процессе вызова сигнал подали несколько объектов, то возвращается наименьший из индексов.
- От WAIT_ABANDONED_0 до (WAIT_ABANDONED_0+nCount-1). Если bWaitAll равно TRUE, возвращаемое значение указывает, что состояние всех опрашиваемых объектов — сигнализирующее, при этом, как минимум, один из объектов представляет собой взаимоисключение, или мьютекс, который не был освобожден владеющим им потоком, а сам поток завершил свою работу. Если значение bWaitAll равно FALSE, возвращаемое значение, уменьшенное на величину WAIT_ABANDONED_0, представляет собой индекс в массиве lpHandles того объекта, который не был освобожден владеющим им и завершившим свою работу потоком.
- WAIT_TIMEOUT. Переданный функции интервал времени истек, а условие, определенное параметром bWaitAll, так и не достигнуто.

Функция WaitForMultipleObjects() проверяет текущее состояние указанных объектов. Если условие, определяемое параметром bWaitAll, не выполнено, вызывающий поток входит в состояние ожидания, однако это ожидание гораздо более эффективно, чем циклический опрос значений переменной. Поток при этом использует очень небольшое процессорное время.

Использование сигналов для синхронизации потоков

Приведенные выше пояснения могут показаться чересчур сухими и переполненными техническими деталями. Поэтому в качестве примера использования рассмотренных функций изменим еще немного демонстрационную программу. В очередной ее версии используем для ожидания завершения дочернего потока вызов функции WaitForSingleObject().

Единственная причина, по которой из программы удалена глобальная переменная-флаг завершения работы — упрощение программы. Кроме того, для простоты используем функцию WaitForSingleObject() и один дочерний поток.

```
// DEMO11_8.CPP - Пример использования функции
//              WaitForSingleObject(...)

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

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

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

DWORD WINAPI Printer_Thread(LPVOID data);

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

DWORD WINAPI Printer_Thread(LPVOID data)
{
   // Этот поток состоит в выводе переданной информации
   // 50 раз с небольшой задержкой

   for (int index=0; index<50; index++)
   {
      printf("%d ",data); // Вывод символа
      Sleep(100);            // Задержка
   } // for index


   // Возвращаем переданные в функцию данные

   return((DWORD)data);

} // Printer_Thread

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

void main(void)
{
   HANDLE thread_handle;  // Дескриптор потока
   DWORD  thread_id;         // Идентификатор потока

   printf("\nStarting threads...\n");

   // Создание потока
   thread_handle = CreateThread(
      NULL,         // безопасность по умолчанию
```

```
    0,                  // стек по умолчанию
    Printer_Thread,     // функция потока
    (LPVOID)1,          // пользовательские данные,
                        // передаваемые потоку
    0,                  // флаг создания
    &thread_id);        // идентификатор потока

  // Выводим 25 символов с тем, чтобы данный вывод
  // завершился раньше, чем работа дочернего потока
  for (int index=0; index<25; index++)
  {
    printf("2 ");
    Sleep(100);

  } // for index

  printf("\nWaiting for thread to terminate\n");

  // Ожидаем завершения дочернего потока
  WaitForSingleObject(thread_handle, INFINITE);

  // Дескриптор потока можно закрывать
  CloseHandle(thread_handle);

  printf("\nAll threads terminated.\n");

} // main
```

Вот как выглядит возможный вывод данной программы:

```
Starting threads...
2 1 2 1 2 1 2 2 1 1 2 1 2 1 2 1 2 2 1 2 1 2 1 2 1 2 2
1 1 2 1 2 1 2 1 2 2 1 2 1 2 1 2 1 2 2 1 1 2 1 2 1 2 1
Waiting for thread to terminate
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
All threads terminated.
```

Программа очень проста. Как обычно, вы создаете дочерний поток, а затем входите в цикл вывода символа на экран. Когда цикл завершается, вызывается функция WaitForSingleObject(). Если вы запустите эту программу при активном SYSMON.EXE, то воочию убедитесь в очень малой загрузке процессора при использовании функцииWaitForSingleObject().

Прежде чем перейти к следующему примеру и использованию нескольких потоков, добавим пару слов об использовании функции WaitForSingleObject(). Допустим, вы хотите узнать состояние некоторого потока в определенный момент, но не хотите ожидать его завершения. Это можно осуществить при помощи вызова функции с передачей ей нулевого интервала времени.

```
//... некоторый код

DWORD state = WaitForSingleObject(thread_handle,0);

if (state == WAIT_OBJECT_0)
{
  // Сигнализирующее состояние - поток завершен
```

```
}
else if (state == WAIT_TIMEOUT)
{
  // Поток все еще работает
}

//... некоторый код
```

Как видите, все достаточно просто. Этот метод проверки завершенности работы определенного потока и использование глобального флага завершения обеспечивают достаточно надежный и эффективный способ завершения работы потоков.

Ожидание нескольких объектов

Итак, рассмотрение функций Wait*() почти закончено — осталось только рассмотреть демонстрационную программу, в которой выполняется ожидание завершения нескольких объектов. Все, что для этого нужно, — создать массив дескрипторов потоков и передать его функции WaitForMultipleObjects().

Демонстрационная программа DEMO11_9.CPP очень похожа на программу DEMO11_8.CPP, с тем отличием, что в ней создается несколько дочерних потоков, завершения которых затем ожидает основной поток. В этой программе также не используется глобальный флаг завершения потоков — каждый дочерний поток просто выполняет свой цикл вывода и завершает работу.

```
// DEMO11_9.CPP - Пример использования функции
//          WaitForMultipleObjects(...)

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

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

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

#define MAX_THREADS 3

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

DWORD WINAPI Printer_Thread(LPVOID data);

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

DWORD WINAPI Printer_Thread(LPVOID data)
{
  // Этот поток состоит в выводе переданной информации
```

```
   // 50 раз с небольшой задержкой

   for (int index=0; index<50; index++)
   {
      printf("%d ",(int)data+1); // Вывод символа
      Sleep(100);        // Задержка
   } // for index


   // Возвращаем переданные в функцию данные

   return((DWORD)data);

} // Printer_Thread

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

void main(void)
{
   HANDLE thread_handle[MAX_THREADS]; // Дескрипторы потоков
   DWORD  thread_id[MAX_THREADS];        // Идентификаторы потоков

   printf("\nStarting threads...\n");

   // Создание потоков
   for (int index=0; index<MAX_THREADS; index++)
   {
      thread_handle[index] = CreateThread(
         NULL,                  // Безопасность по умолчанию
         0,                     // Стек по умолчанию
         Printer_Thread,        // Функция потока
         (LPVOID)index,         // Пользовательские данные,
                                // передаваемые потоку
         0,                     // Флаг создания
         &thread_id[index]);  // Идентификатор потока

   // Выводим 25 символов с тем, чтобы данный вывод
   // завершился раньше, чем работа дочерних потоков
   for (int index=0; index<25; index++)
   {
      printf("4 ");
      Sleep(100);

   } // for index

   // Ожидаем завершения дочерних потоков
   WaitForMultipleObjects(
      MAX_THREADS,   // Количество потоков
      thread_handle,   // Дескрипторы потоков
      TRUE,             // Ожидать все потоки
      INFINITE);        // Ждать без ограничений по времени

   // Дескрипторы потоков можно закрывать
```

```
  for (index=0; index<MAX_THREADS; index++)
    CloseHandle(thread_handle[index]);

  printf("\nAll threads terminated.\n");

} // main
```

Вот как выглядит возможный вывод данной программы:

```
Starting all threads...
4 1 2 3 4 1 2 3 4 4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4
1 2 3 4 1 2 3 4 4 1 2 3 4 1 2 3 4 1 2 3 4 4 1 2 3
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 4 1 2 3 4
1 2 3 4 1 2 3 4 4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
All threads terminated.
```

Все происходит именно так, как предполагалось. Все потоки продолжают выводить свои цифры вместе с основным потоком, который завершает вывод первым и переходит в состояние ожидания завершения всех дочерних потоков. Когда все дочерние потоки завершатся, основной поток возвращается из функции WaitForMultipleObjects() и работа программы завершается.

Многопоточность и DirectX

Теперь, когда вы получили некоторое представление о многопоточности, возникает вопрос о том, как использовать эту возможность в игровых программах и программах, использующих DirectX. Да просто использовать и ничего более! Конечно, следует убедиться, что при компиляции у вас используются именно многопоточные библиотеки. Кроме того, при работе с ресурсами DirectX может возникнуть ряд проблем, связанных с "критическими разделами".

Убедитесь в том, что вами используется глобальная стратегия работы с ресурсами, т.е. если к ресурсу обращается несколько потоков, то при этом не возникает никаких проблем. Например, допустим, что один поток заблокировал некий ресурс, к которому в это время пытается обратиться другой поток; это может привести к проблемам. Для их решения используются *семафоры*, *взаимоисключения* и *критические разделы*. К сожалению, у меня совершенно нет возможности рассматривать здесь их работу, но, к счастью, есть множество отличных книг на эту тему, например *Multithreading Applications in Win32* Джима Бевериджа (Jim Beveridge) и Роберта Вайнера (Robert Weiner).

Вкратце идея такова. Для корректной работы с ресурсами нескольких потоков можно создать переменные, предназначенные для отслеживания использования ресурсов каким-либо потоком. Любой поток, которому требуется доступ к ресурсу, должен подождать, пока этот ресурс не будет освобожден потоком, работающим с ним в настоящий момент. Кстати, здесь может возникнуть проблема, связанная с тем, что компилятор может оптимизировать процесс проверки, создав регистровую копию переменной и тем самым не позволяя отследить ее изменение другим потоком. Этой проблемы можно избежать, описав переменную как volatile.

Однако Win32 предоставляет вместо таких переменных гораздо более эффективные возможности, а именно *семафоры* (по сути, счетчики наподобие глобальных переменных, но реализованные атомарно, т.е. в процессе вызова функции API для работы с семафором его состояние не может быть изменено другим потоком), *взаимоисключения*

(бинарные семафоры, которые разрешают работу с ресурсом только одному потоку) и *критические разделы* (части кода, в которых одновременно может находиться только один поток). Если же потоки не зависят друг от друга (в том числе не используют одни и те же ресурсы), то никаких специальных мер при их написании принимать не нужно.

В качестве примера приложения DirectX, использующего многопоточность, на прилагаемом компакт-диске представлена программа DEMO11_10.CPP (ее 16-битовая версия находится в файле DEMO11_10_16B.CPP). В этой программе создается ряд объектов, перемещающихся по экрану. В 32-битовой версии потоки используются для изменения цвета объектов, а в 16-битовой — для их перемещения. Единственное замечание по компиляции данной программы: не забудьте подключить .LIB-файлы DirectX.

Однако, если у вас много потоков, которые вызывают одни и те же функции, может возникнуть проблема реентерабельности. Реентерабельная функция должна хранить информацию о состоянии и не может использовать глобальные переменные, которые могут быть изменены в результате работы другого потока.

Кроме того, если вы используете потоки для анимации объектов DirectX, вам придется заботиться о согласованности поверхностей, вопросах синхронизации и т.п. Поэтому я бы рекомендовал ограничить применение потоков для выполнения работы, которая очень мало зависит от других частей программы, находится в собственном "пространстве состояний" и не должна выполняться со строго фиксированной скоростью.

Дополнительные вопросы многопоточности

Пожалуй, лучше всего в изложении многопоточности остановиться на этом месте и ни слова не говорить об условиях гонки, взаимоблокировках, критических разделах, взаимоисключениях, семафорах и прочих вещах, вызывающих головную боль. Знание всего перечисленного помогает в написании таких многопоточных программ, в которых потоки сотрудничают, а не мешают друг другу. Но даже без знания всех тонкостей, ориентируясь только на здравый смысл и помня, что один поток может прервать работу другого в любой самый неподходящий момент, можно писать безопасные и эффективные многопоточные программы. Будьте особенно внимательны при совместном использовании разными потоками одних и тех же ресурсов.

К сожалению, размер книги не позволяет рассмотреть не только множество вопросов многопоточности, но и многие функции API, такие, как ExitThread() или GetThreadExitCode(), с которыми вам придется самостоятельно знакомиться с помощью справочной системы Win32.

Резюме

Правда, эта глава была не слишком сложной? Так, винегрет из разнообразной информации... На самом деле здесь были затронуты весьма серьезные темы, представляющие собой основу для дальнейшего чтения книги, — управление памятью, структуры данных, рекурсия и многое другое.

И хотя изложенное здесь может показаться довольно сложным, при разработке игр все это может понадобиться в любой момент. Например, при разработке системы искусственного интеллекта, о чем и пойдет речь в следующей главе.

ГЛАВА 12

Искусственный интеллект в игровых программах

Эта глава должна ответить на множество вопросов об искусственном интеллекте. Изучив представленный в ней материал, вы сможете написать код и алгоритмы для создания персонажей игры, осмысленно ведущих себя и выполняющих почти все, что вы пожелаете. Вот что рассматривается в данной главе.

- Азы искусственного интеллекта
- Простые детерминированные алгоритмы
- Шаблоны и сценарии
- Поведение и состояние
- Запоминание и обучение
- Деревья планирования и принятия решений
- Поиск путей
- Языки сценариев
- Основы нейронных сетей
- Генетические алгоритмы
- Нечеткая логика

Азы искусственного интеллекта

В академическом смысле термин “искусственный интеллект” (ИИ) означает аппаратное или программное обеспечение, которое позволяет компьютеру “думать”, т.е. обрабатывать информацию подобно человеку. Применение систем ИИ началось не так давно, но растет с экспоненциальной скоростью!

Технологии ИИ стали возможны благодаря *искусственным нейронным сетям*, *генетическим алгоритмам* и *нечеткой логике*. Нейронные сети представляют собой грубую модель человеческого мозга, а генетические алгоритмы — множество методов и гипотез, использующихся для эволюции программной системы, основанной на биологических парадигмах. Нечеткая логика является теорией множеств, основанной на нечетких гипотезах.

Звучит непонятно и странно? Конечно. Но все эти технологии приближаются к реальному миру и работают все лучше и лучше. В конце концов, еще не так давно клонирование было всего лишь фантастикой, а сегодня это научный факт.

Спускаясь с заоблачных высот на грешную Землю, скажем сразу: мы не собираемся создавать реальный искусственный интеллект для игр. Вместо этого мы будем рассматривать упрощенные методы, используемые программистами для создания разумных персонажей, вернее, *выглядящих* разумными. Как ни странно, но очень многие программисты знакомы с вопросами ИИ лишь поверхностно, и в результате получаются игры, в которых великолепная трехмерная графика сочетается с полным идиотизмом персонажей.

После того как вы прочитаете главу и испытаете все демонстрационные программы, запомните, что все описанные здесь технологии остаются только технологиями и не более. Важно не то, верен или неверен способ решения задачи, важно, чтобы задача была решена. В конце концов, если управляемый компьютером танк может выехать из засады и выстрелить вам в спину, значит, вы получили желаемое. Если нет — следует продолжать работу.

Независимо от того, насколько примитивны используемые вашим искусственным интеллектом технологии, игрок всегда будет наделять своих компьютерных противников индивидуальностью. Самую главную часть работы игрок сделает за вас — ведь он будет верить, что его противники *действительно* думают, планируют, хитрят, по крайней мере внешне это будет *выглядеть* именно так. А большего нам и не требуется, не так ли?

Детерминированные алгоритмы искусственного интеллекта

Детерминированные алгоритмы означают предопределенное и заранее запрограммированное поведение объектов. Например, если вы рассмотрите систему ИИ в демонстрационной игре Asteroids из главы 8, “Растеризация векторов и двухмерные преобразования” (рис. 12.1), то увидите, что он крайне прост.

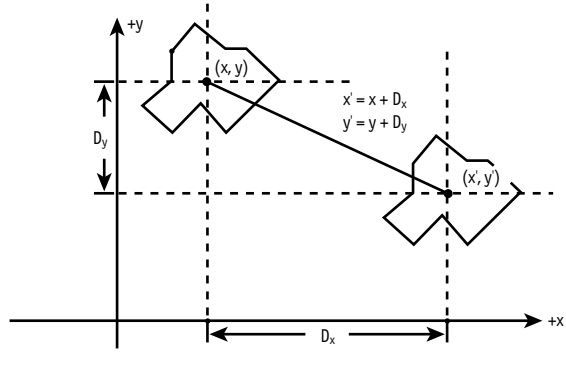


Рис. 12.1. Искусственный интеллект игры Asteroids

Искусственный интеллект создает астероид, который затем отправляет в полет в случайном направлении со случайной скоростью. Вся математика ИИ сводится к двум строкам программы:

```
asteroid_x += asteroid_x_velocity;
asteroid_y += asteroid_y_velocity;
```

У астероида одна цель: двигаться по своему курсу. Все. Поэтому искусственный интеллект астероида предельно прост — ему не требуется обрабатывать какой-то внешний ввод информации, изменять курс и т.п. Поведение астероидов полностью детерминировано и предсказуемо.

Именно такие, простые и предсказуемые алгоритмы ИИ и составляют первый класс алгоритмов, которые я хочу рассмотреть. В этом классе имеется ряд технологий, которые были разработаны еще во времена игр типа PacMan.

Случайное движение

В описанный выше алгоритм достаточно внести минимальные изменения, чтобы получить хаотическое перемещение объекта (или изменение его свойств), как показано на рис. 12.2.

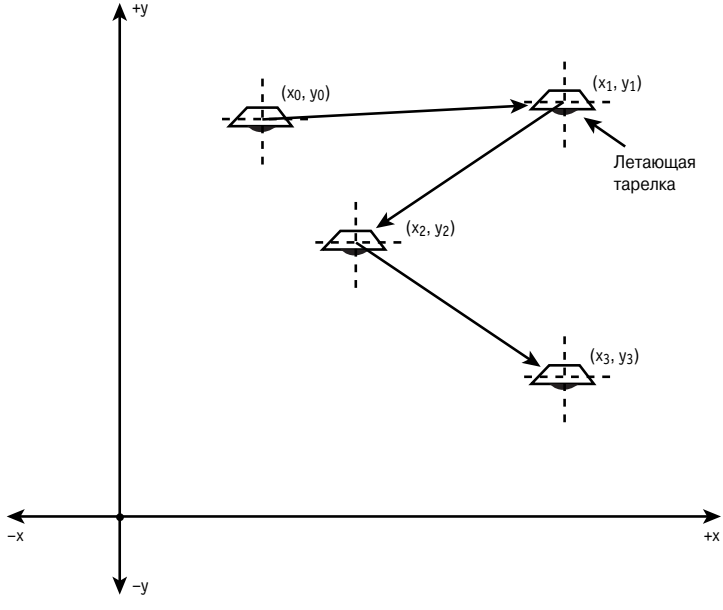


Рис. 12.2. Случайное движение объекта

Например, если вы хотите смоделировать броуновское движение атома или какой-то другой объект с хаотичным движением, можно поступить так. Сначала случайным образом изменяется скорость объекта:

```
fly_x_velocity = -8 + rand() % 16;
fly_y_velocity = -8 + rand() % 16;
```

Затем объект некоторое время движется с вычисленной скоростью:

```
// Движение с заданной скоростью в течение 10 циклов
for(int fly_count = 0; fly_count < 10; ++fly_count)
{
  fly_x += fly_x_velocity;
  fly_y += fly_y_velocity;
}

// ... Вычисление новой скорости объекта ...
```

В данном примере сначала выбирается случайное направление движения и скорость, после чего в течение 10 циклов объект перемещается с использованием выбранного вектора скорости. Затем выбирается новое направление и скорость, и цикл повторяется. Очевидно, что в данный алгоритм можно внести дополнительный уровень хаотичности, например выбирая некоторое случайное число вместо 10 циклов перемещения. Кроме того, можно отдать предпочтение выбору одних направлений перед другими (тем самым, например, можно сымитировать наличие ветра).

Данный пример показывает, что можно добиться достаточно интересных результатов при помощи очень небольшого кода. Чтобы увидеть это воочию на экране, можно запустить демонстрационные программы DEMO12_1.EXE (или 16-битовую версию DEMO12_1_16B.EXE), которые вместе с исходными текстами находятся на прилагаемом компакт-диске.

Случайное движение представляет собой важную часть поведенческой модели интеллектуальных объектов игры. Небольшое примечание об интеллектуальности: я живу в Кремниевой Долине, и могу засвидетельствовать, что водители здесь часто ведут себя в соответствии с описанным алгоритмом: хаотично меняют полосы движения, а иногда и просто двигаются по встречной полосе...

Алгоритм следования за объектом

Хотя случайное перемещение и является совершенно непредсказуемым, обычно от него мало толку. Следующим шагом в развитии ИИ являются алгоритмы, которые учитывают состояние внешней среды и реагируют на него. В качестве примера я выбрал алгоритм следования за объектом (tracking algorithm). Искусственный интеллект на основе информации о положении некоторого (отслеживаемого) объекта изменяет траекторию нашего объекта таким образом, чтобы он двигался по направлению к отслеживаемому.

Движение может быть направлено как непосредственно на отслеживаемый объект, так и (в более реалистичной модели) по некоторой кривой, направленной к объекту (наподобие траектории самонаводящейся ракеты), что показано на рис. 12.3.

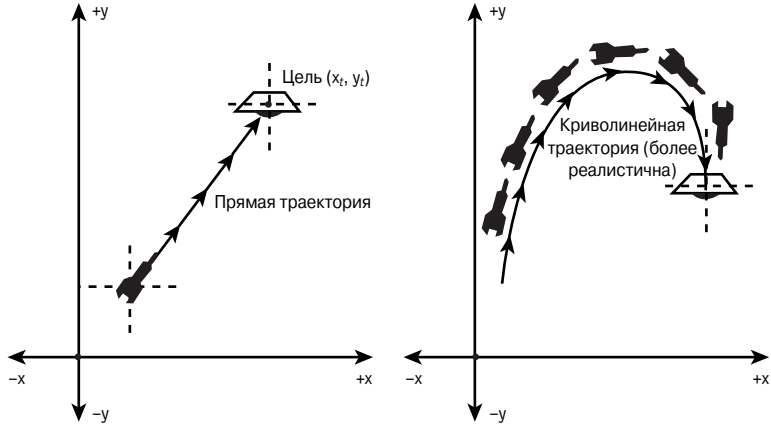


Рис. 12.3. Методы отслеживания перемещения

В качестве грубого примера следования за объектом можно привести следующий алгоритм:

```
// Координаты игрока - player_x, player_y
// Координаты персонажа - monster_x, monster_y
```

```
// Проверка относительного размещения по оси X
if (player_x > monster_x) ++monster_x;
if (player_x < monster_x) --monster_x;

// Проверка относительного размещения по оси Y
if (player_y > monster_y) ++monster_y;
if (player_y < monster_y) --monster_y;
```

Если вы используете этот алгоритм ИИ в простейшей демонстрационной программе, то получите эдакого Терминатора, постоянно и неуклонно направляющегося к цели. Приведенный код очень прост, но весьма эффективен. Кстати, практически аналогичный алгоритм использован в игре PacMan (конечно, с определенными изменениями, позволяющими обходить препятствия и двигаться только по прямым линиям). В качестве примера работы данного алгоритма рассмотрите находящиеся на прилагаемом компакт-диске программу DEMO12_2.CPP и ее 16-битовую версию DEMO12_2_16B.CPP. В этой игре, управляя приведением с помощью клавиш перемещения курсора, вы можете посмотреть, как послушно следует за ним летучая мышь.

Этот алгоритм следования неплох, но представляется не очень интеллектуальным; более естественным путем было бы движение с учетом направления вектора от центра нашего объекта к центру объекта, за которым мы следуем (рис. 12.4).

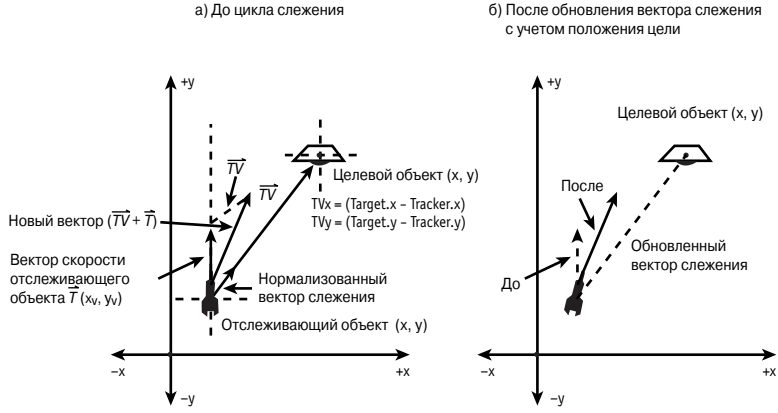


Рис. 12.4. Следование за объектом с использованием вектора слежения

Данный алгоритм работает следующим образом. Пусть управляемый искусственным интеллектом объект называется tracker и имеет свойства

```
Координаты: (tracker.x,  tracker.y)
Скорость  : (tracker.xv, tracker.yv)
```

и пусть отслеживаемым объектом будет target со свойствами

```
Координаты: (target.x,  target.y)
Скорость  : (target.xv, target.yv)
```

Используя данные определения, алгоритм можно описать следующим образом.

1. Вычисляем вектор **TV** от объекта до цели:

   TV = (target.x-tracker.x, target.y-tracker.y) = (tvx, tvy). Затем нормализуем его, т.е. делим на длину. Обозначим нормализованный вектор как **TV***:

   TV* = (tvx,tvy)/sqrt(tvx*tvx+tvy*tvy).

2. Изменяем текущий вектор скорости отслеживающего объекта, добавляя к нему нормализованный вектор **TV***, умноженный на константу rate:

```
tracker.xv += rate*tvx;
tracker.yv += rate*tvy;
```

Если взять значение rate, превышающее 1.0, то изменение траектории отслеживающего объекта будет более крутым, в результате цель будет настигнута быстрее.

3. После изменения скорости объекта следует проверить, не превышает ли она максимального допустимого значения. Если это так, значение скорости следует снизить до приемлемого.

```
// Получаем значение скорости объекта
tspeed =
  sqrt(tracker.xxv*tracker.xv + tracker.yv*tracker.yv);

// Скорость слишком велика?
if (tspeed > MAX_SPEED)
{
  // Уменьшаем вектор скорости
  tracker.xv *= 0.75;
  tracker.yv *= 0.75;
} // if
```

Можно использовать и другие значения множителя, например 0.5 или 0.9, это не так важно. Можно даже поступить совсем честно и просто вычислить необходимое значение коэффициента, которое обеспечит необходимое максимальное значение скорости после умножения.

Вот как реализован этот алгоритм в демонстрационной программе DEMO12_3.CPP, откуда и взят приведенный ниже фрагмент кода:

```
// Алгоритм следования

// Вычисляем вектор, направленный к игроку
float vx = player_x - mines[index].varsI[INDEX_WORLD_X];
float vy = player_y - mines[index].varsI[INDEX_WORLD_Y];

// Нормализуем вектор
float length = Fast_Distance_2D(vx,vy);


// Изменяем траекторию при относительной близости
if (length < MIN_MINE_TRACKING_DIST)
  {
  vx=mine_tracking_rate*vx/length;
  vy=mine_tracking_rate*vy/length;

  // Добавляем вектор к текущей скорости
  mines[index].xv+=vx;
  mines[index].yv+=vy;

  // Вносим немного случайности
  if ((rand()%10)==1)
    {
```

```
    vx = RAND_RANGE(-1,1);
    vy = RAND_RANGE(-1,1);
    mines[index].xv+=vx;
    mines[index].yv+=vy;
    } // if

  // Проверяем значение скорости
  // и при необходимости уменьшаем его
  length = Fast_Distance_2D(mines[index].xv,
                    mines[index].yv);

  if (length > MAX_MINE_VELOCITY)
    {
    // Уменьшение скорости
    mines[index].xv*=0.75;
    mines[index].yv*=0.75;

    } // if

  } // if
else
  {
  // Добавляем случайную составляющую к скорости
  if ((rand()%30)==1)
    {
    vx = RAND_RANGE(-2,2);
    vy = RAND_RANGE(-2,2);

    mines[index].xv+=vx;
    mines[index].yv+=vy;

    // Проверяем значение скорости
    // и при необходимости уменьшаем его
    length = Fast_Distance_2D(mines[index].xv,
                      mines[index].yv);

    if (length > MAX_MINE_VELOCITY)
      {
      // Уменьшение скорости
      mines[index].xv*=0.75;
      mines[index].yv*=0.75;

      } // if

    } // if

  } // else
```

Понятно, что данный фрагмент взят из цикла, в котором обрабатывается движение целого ряда объектов, однако для понимания сути алгоритма это не имеет никакого значения. Этот фрагмент представляет собой четкую реализацию описанного ранее алгоритма, хотя здесь есть свои тонкости, на которые я бы хотел обратить ваше внимание. Например, в приведенном фрагменте есть проверка того, насколько далеко следящий

объект (в нашем случае летающая в космическом пространстве мина) находится от цели. Если это расстояние превышает некоторое заранее заданное, мина продолжает свой путь, не замечая цели и лишь слегка изменяя свой курс случайным образом. Кроме того, даже когда мина гонится за игроком, в ее движение вносится элемент случайности, что придает процессу погони еще больший реализм.

В случае перемещения в среде, взаимодействующей с объектом (воздух, вода, гравитационное поле и т.п.), следует изменять используемую физическую модель.

Описанный алгоритм, как уже упоминалось, применен в демонстрационной программе DEMO12_3.CPP, представленной на прилагаемом компакт-диске. В игре вы можете перемещать небольшой корабль в замкнутой вселенной. В ее пределах имеется некоторое количество мин, которые способны гоняться за кораблем с использованием рассмотренного алгоритма. Управляющие клавиши игры:

Клавиши перемещения курсора	Управление перемещением корабля
<Ctrl>	Стрельба
<+>/<->	Изменение степени отслеживания перемещения корабля минами
<H>	Вкл./Выкл. вывода информации об игре на экране
<S>	Вкл./Выкл. экрана сканера

При игре обратите внимание на изменение поведения мин при изменении степени отслеживания перемещения корабля.

Приведенная демонстрационная программа — хороший пример небольшой игры, так что постарайтесь изучить ее как можно лучше.

СОВЕТ

Поскольку я использую для вывода текста средства GDI, вывод текста существенно замедляет работу игры. Это задумано специально: я хотел, чтобы вы обратили на это внимание. В реальных играх следует использовать собственные быстрые системы вывода текста.

Алгоритм уклонения от объекта

Теперь рассмотрим алгоритм ИИ, который позволяет объекту убегать от вас. Этот алгоритм ничуть не сложнее алгоритма следования за объектом; по сути, вам просто нужно изменить знак скорости на противоположный.

```
// Координаты игрока - player_x, player_y
// Координаты персонажа - monster_x, monster_y

// Проверка относительного размещения по оси X
if (player_x > monster_x) --monster_x;
if (player_x < monster_x) ++monster_x;

// Проверка относительного размещения по оси Y
if (player_y > monster_y) --monster_y;
if (player_y < monster_y) ++monster_y;
```

НА ЗАМЕТКУ

Обратите внимание, что при проверке относительного положения объектов не используется равенство их координат (сравнение ==). Это связано с тем, что я не хочу перемещать объект при совпадении координат. Однако вам никто не мешает использовать проверку на равенство и выполнять при этом какие-то необходимые действия.

Дочитав главу до этого места, вы уже в состоянии создать впечатляющую систему ИИ, обеспечивающую хаотичное движение, следование за объектом и уклонение от него.

Этого вполне достаточно для разработки игры типа PacMan. Возможно, вас это и не очень впечатляет, но, думаю, продажа ста миллионов копий игры — это далеко не самый плохой результат!

Для того чтобы посмотреть работу алгоритма уклонения от объекта в действии, воспользуйтесь имеющейся на прилагаемом компакт-диске демонстрационной программой DEMO12_4.CPP (и ее 16-битовой версией DEMO12_4_16B.CPP). Эта программа очень похожа на программу DEMO12_2.CPP, но алгоритм следования за объектом заменен в ней алгоритмом уклонения от него.

А теперь перейдем к шаблонам.

Шаблоны и сценарии

Детерминированные алгоритмы вполне достаточны для решения множества задач, тем не менее довольно часто требуется создать объект, который действует по некоторому сценарию. Например, вот приблизительный сценарий ваших действий при поездке в автомобиле.

1. Вынуть ключи из кармана.
2. Открыть дверцу машины с помощью ключа.
3. Сесть в автомобиль.
4. Закрыть дверцу.
5. Вставить ключ в замок зажигания.
6. Повернуть ключ.
7. Завести двигатель.

Словом, следует выполнить целый ряд последовательных действий, повторяющихся каждый раз при необходимости куда-то ехать. Конечно, если что-то пойдет не так, последовательность может быть изменена: например, если вы оставили автомобиль в гараже не запертым или если у вас сел аккумулятор. Шаблоны представляют собой важную часть интеллектуального поведения, включая поведение, присущее венцу творения на нашей планете.

Шаблоны

Создание шаблона для объекта игры может оказаться как сложным, так и очень простым, в зависимости от того, для какого объекта создается шаблон. Например, шаблон для управления движением реализуется очень просто. Представим, что мы разрабатываем игру типа Phoenix или Galaxian. Чужие двигаются по определенной траектории и в некоторый момент атакуют вас, следуя некоторому шаблону атаки. Искусственный интеллект такого рода может использовать множество различных технологий, но лично мне простейшим представляется способ, основанный на трансляции инструкций движения, как показано на рис. 12.5.

Каждый шаблон движения хранится в виде последовательности инструкций, например, приведенных в табл. 12.1

Вместе с каждой инструкцией могут использоваться дополнительные данные, уточняющие ее, например указывающие, на какое расстояние следует переместиться. В результате формат инструкции языка шаблона можно записать следующим образом:

```
ИНСТРУКЦИЯ  ОПЕРАНД
```

ИНСТРУКЦИЯ представляет собой один из кодов из приведенного выше списка, а ОПЕРАНД — числовое значение, которое позволяет уточнить вызываемое инструкцией по-

ведение объекта. Используя этот простейший формат инструкции, вы можете создать программу (последовательность инструкций), определяющую шаблон. Затем вы создаете транслятор, который в процессе игры выполняет разбор шаблона и соответственно управляет действиями персонажа игры.

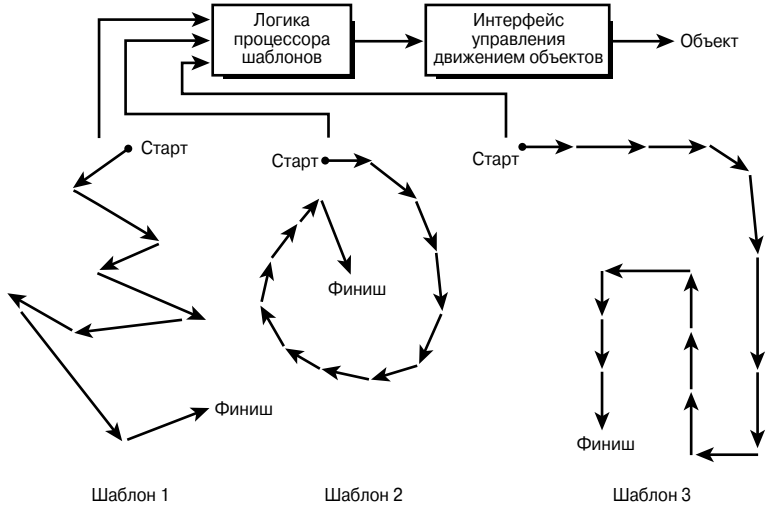


Рис. 12.5. Работа с шаблонами

Таблица 12.1. Гипотетический набор инструкций шаблона

Инструкция	*Значение*
GO_FORWARD	1
GO_BACKWARD	2
TURN_RIGHT_90	3
TURN_LEFN_90	4
SELECT_RANDOM_DIRECTION	5
STOP	6

Пусть, например, в нашем языке шаблона инструкция представляет собой пару чисел, первое из которых определяет действие, а второе — его продолжительность в циклах. Создадим тривиальный шаблон, который соответствует движению по квадрату, как показано на рис. 12.6.

```
int num_instructions = 9; // Количество инструкций в шаблоне

// Программа шаблона
int square_stop_spin[] = {
   1, 30,   3, 1,   // Движение вперед и поворот вправо
   1, 30,   3, 1,   // Движение вперед и поворот вправо
   1, 30,   3, 1,   // Движение вперед и поворот вправо
   1, 30,            // Движение вперед; квадрат замкнут
```

```
6, 60,          // Останов на 60 циклов
4, 8 };         // Поворот в течение 8 циклов
```

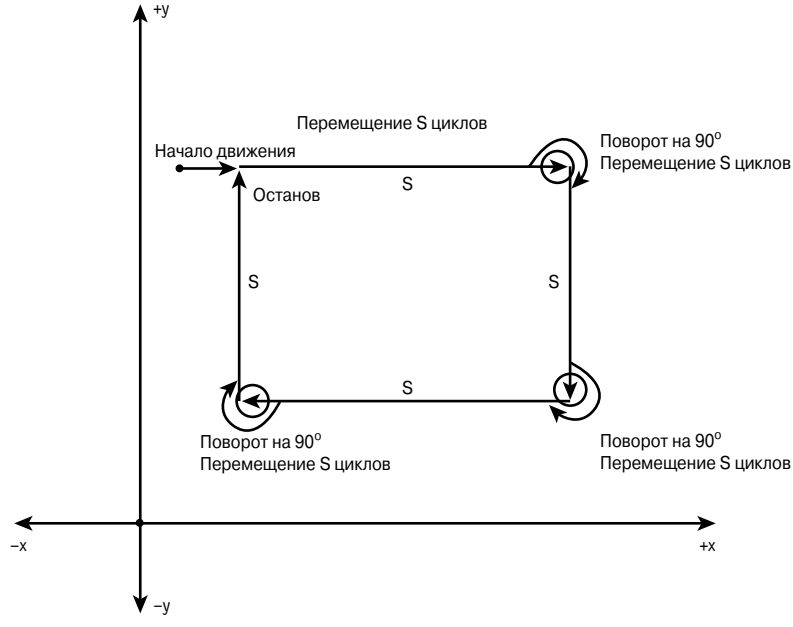


Рис. 12.6. Шаблон движения по квадрату

Все, что требуется для трансляции этой программы, — большой оператор switch(), интерпретирующий каждую инструкцию и передающий соответствующие указания объекту. Приведем пример такого транслятора.

```
// Указывает на первую инструкцию.
// Каждая инструкция состоит из двух целых чисел.
int instruction_ptr = 0;

// Получаем число циклов
int cycles = square_stop_spin[instruction_ptr + 1];

// Обрабатываем инструкцию
switch(square_stop_spin[instruction_ptr])
{
  case GO_FORWARD:     // Перемещаем объект вперед
    break;
  case GO_BACKWARD:    // Перемещаем объект назад
    break;
  case TURN_RIGHT_90:  // Поворот объекта вправо
    break;
  case TURN_LEFT_90:   // Поворот объекта влево
    break;
  case SELECT_RANDOM_DIRECTION:
                       // Выбор случайного направления
    break;
  case STOP:           // Остановить объект
    break;
```

```
} // switch

// Перемещаем указатель инструкций (на 2 числа)
instruction_ptr += 2;

// Проверяем, не добрались ли до конца программы
if (instruction_ptr > num_instructions * 2)
{
  // Программа завершена
}
```

Конечно, к этому фрагменту нужно добавить код, реализующий движение объекта, а также счетчик циклов. Однако общая идея должна быть понятна.

У всех шаблонов есть одна тонкость — *разумность движения*. Объект в игре твердо следует заданному шаблону, что в изменяющихся условиях игры не всегда благоразумно. Поэтому в вашей системе ИИ должна быть обратная связь, выясняющая, не происходит ли нечто некорректное, неразумное или вовсе невозможное, и позволяющая вовремя перейти к другому шаблону или стратегии в целом, как показано на рис. 12.7.

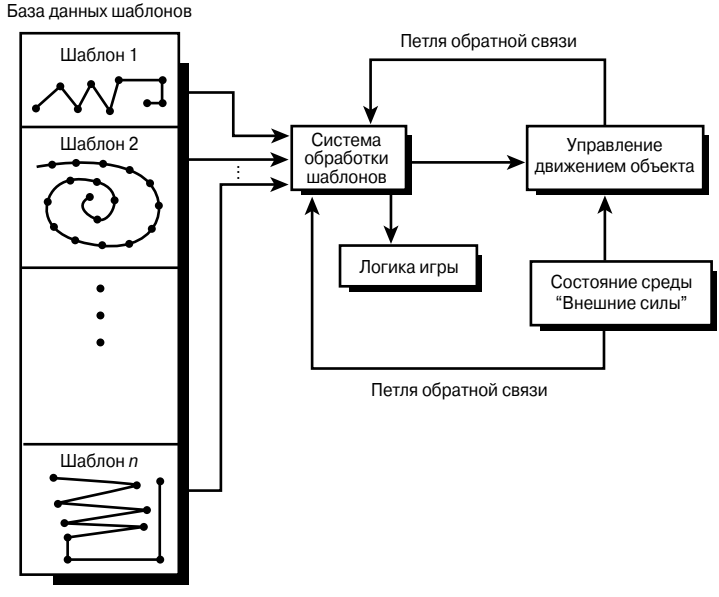


Рис. 12.7. Искусственный интеллект с обратной связью

НА ЗАМЕТКУ

Разумеется, вы можете использовать не массив, а те структуры данных, которые в большей степени подходят для вашей задачи. Например, можно воспользоваться классом или структурой, которые содержат список записей в формате [ИНСТРУКЦИЯ, ОПЕРАНД] вместе с количеством инструкций. При таком способе хранения шаблонов можно легко создать их массив, а затем выбирать из него необходимый шаблон для трансляции и выполнения.

А теперь хочу обратить ваше внимание на то, какая мощь заключена в шаблонах. Используя их, вы можете записать сотни траекторий. Сложные перемещения объектов, которые практически невозможно осуществить с помощью ИИ другого типа за приемлемое время, создаются с помощью простейшего инструментария (который ничего не стоит написать для каждого конкретного случая) всего за несколько минут, после чего записы-

ваются в файл и воспроизводятся в игре. Применение шаблонов способно сделать так, что персонажи вашей игры будут выглядеть и поступать исключительно разумно — ничуть не хуже самого игрока. Именно этот метод был использован в таких играх, как Dead of Alive, Tekken, Soul Blade, Mortal Kombat и др.

Кроме того, применение шаблонов не ограничивается только шаблонами движения. Вы можете использовать шаблоны и для других целей, например выбора вооружения, управления анимацией и т.п. — список возможных применений ограничивается только вашей фантазией.

На прилагаемом компакт-диске находится демонстрационная программа DEMO12_5.CPP (и ее 16-битовая версия DEMO12_5_16B.CPP), использующая для перемещения персонажа целый ряд шаблонов.

Шаблоны с обработкой условий

Шаблоны — достаточно мощная, однако полностью детерминированная технология. А это означает, что как только игрок запомнит шаблон, он становится бесполезен: игрок всегда знает, что произойдет дальше. Решение этой проблемы (как и ряда других проблем, связанных с шаблонами) состоит в добавлении к логике обработки шаблонов условных операторов, что позволяет учитывать состояние игры и действия игрока. Абстрагированная схема работы такого шаблона показана на рис. 12.8.

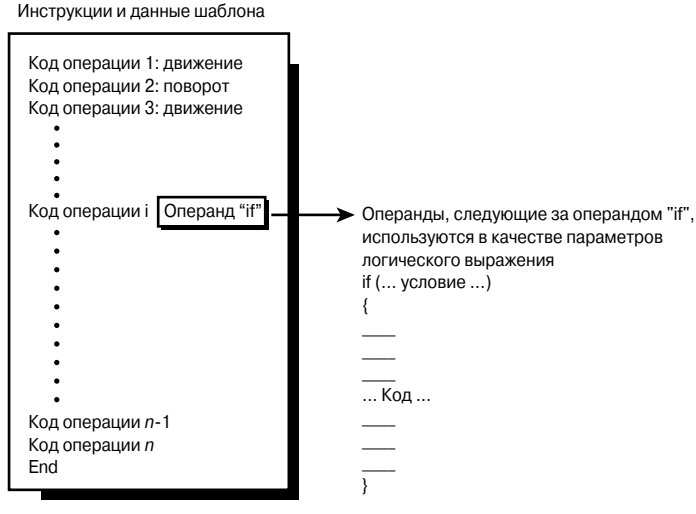


Рис. 12.8. Шаблоны с условными операторами

Шаблоны с условными операторами дают вам еще один уровень контроля над системой ИИ, позволяя создавать шаблоны с ветвлением. Например, вы можете добавить в язык шаблонов новую инструкцию

```
TEST_DISTANCE  7
```

Она проверяет расстояние от игрока до объекта, выполняющего шаблон. Если дистанция слишком велика (или, напротив, мала либо выполняется еще какое-то условие, налагаемое на расстояние от игрока до объекта), то система ИИ может изменить свое поведение.

Вы можете поместить инструкцию TEST_DISTANCE в шаблон точно так же, как и любую другую.

```
TURN_RIGHT_90, GO_FORWARD, ..., TEST_DISTANCE, ...
TURN_LEFT_90, ..., TEST_DISTANCE, ..., GO_BACKWARD
```

При выполнении шаблона каждая встречающаяся инструкция TEST_DISTANCE заставляет систему ИИ определить расстояние от игрока до объекта и сравнить его с расстоянием, заданным в качестве операнда инструкции TEST_DISTANCE. Если расстояние слишком велико, искусственный интеллект может переключиться, например, на выполнение другого шаблона, который целенаправленно будет снижать указанное расстояние. Взгляните на следующий код:

```
if (instruction_stream[instruction_ptr++] == TEST_DISTANCE)
{
  // Получаем значение операнда инструкции TEST_DISTANCE
  // и расстояние от игрока до объекта и сравниваем их

  int min_distance = instruction_stream[instruction_ptr++];
  if (Distance(player,object) > min_distance)
  {
    // Изменяем внутреннее состояние системы
    // искусственного интеллекта...
    ai_state = TRACK_PLAYER;

    // ...или переключаемся на выполнение другого шаблона
  } // if
} // if
```

Сложность выполняемых при работе шаблона проверок может быть любой. Кроме того, возможно создание шаблонов, применяемых при выполнении условий, "на лету", в процессе игры. В частности, одним из примеров может служить подражание игроку: вы отслеживаете, что именно делал игрок, чтобы уничтожить персонаж игры, и затем используете тактику игрока против него самого!

В заключение замечу, что описанная технология используется как в ряде спортивных игр, таких, как футбол, бейсбол или хоккей, так и в стратегических играх. Такие шаблоны, обеспечивая предсказуемость персонажа игры, позволяют ему при необходимости полностью сменить тактику и стратегию.

На прилагаемом компакт-диске находится демонстрационная программа DEMO12_6.CPP, в которой игрок с помощью клавиш перемещения курсора управляет движением летучей мыши, а система ИИ руководит перемещениями скелета. Скелет следует случайно выбранному шаблону до тех пор, пока он не очень удалился от летучей мыши. Если он удалится слишком далеко, ему становится одиноко, происходит переключение стратегий и он стремится подойти к вам поближе, чтобы насладиться обществом... Правда, неплохое эмоциональное обоснование работы 100 строк кода? (Но разве с точки зрения стороннего наблюдателя все выглядит не именно так?)

Моделирование систем с состояниями

В этом разделе рассматриваются *конечные автоматы* (которые именуют также машинами состояний). Моя цель — формализовать применение конечных автоматов при разработке систем ИИ в игровых программах.

Для создания устойчивого конечного автомата требуется следующее:

- разумное количество состояний, каждое из которых представляет различные цели или мотивы;
- входная информация для конечного автомата, такая, как состояние среды и других объектов в ней.

Требование разумного количества состояний достаточно просто и понятно. У нас, людей, имеются сотни, если не тысячи, эмоциональных состояний, и в каждом из них — множество подсостояний.

А персонаж игры должен всего лишь выглядеть разумным, так что и состояний у него не должно быть чересчур много. Например, простой персонаж игры может иметь несколько состояний.

- Состояние 1: Двигаться вперед.
- Состояние 2: Двигаться назад.
- Состояние 3: Поворот.
- Состояние 4: Останов.
- Состояние 5: Стрельба.
- Состояние 6: Погоня за игроком.
- Состояние 7: Бегство от игрока.

Состояния 1–4 просты и очевидны, однако для корректного моделирования поведения персонажа в состояниях 5–7 могут потребоваться дополнительные подсостояния. Например, погоня за игроком может включать в себя движение вперед и повороты. Взгляните на рис. 12.9, где схематично проиллюстрирована концепция подсостояний. Однако не следует считать, что подсостояния могут быть основаны только на уже имеющихся состояниях; подсостояния могут быть искусственно введены для каждого из рассматриваемых состояний.

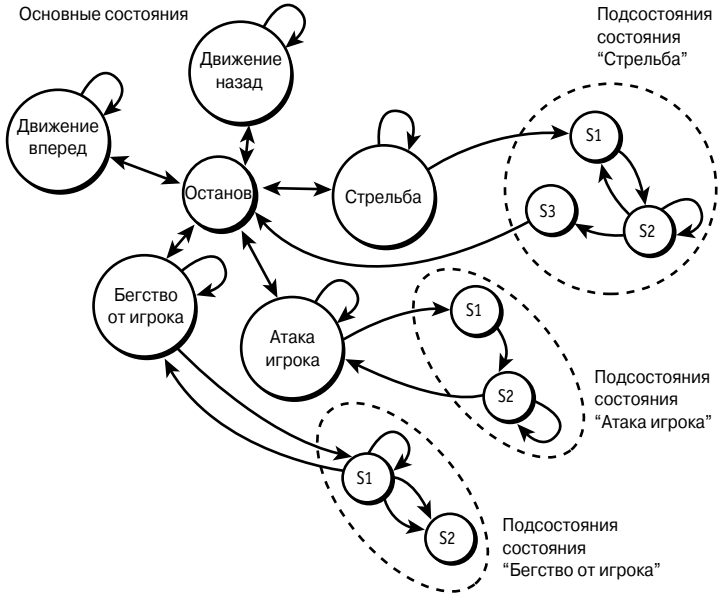


Рис. 12.9. Конечный автомат с подсостояниями

Вывод из этого обсуждения таков: состояний не должно быть слишком много, иначе будет просто невозможно написать соответствующую программу, и в то же время не должно быть слишком мало, иначе объект не будет казаться достаточно разумным. Если у объекта только два состояния, например стоять и двигаться вперед, то трудно добиться

впечатления разумности такого объекта. Состояний должно быть ровно столько, сколько требуется для получения желаемого эффекта разумности, и ни одним больше. Не очень точная подсказка, не правда ли? :)

Что касается второго требования, то вам необходимо обеспечить получение информации от других объектов в игре, от игрока и игровой среды. Если объект просто входит в некоторое состояние и находится в нем до конца игры, это не искусственный интеллект, а искусственная глупость. Состояние могло быть выбрано вполне разумно, но это было 100 миллисекунд тому назад. С того времени игровая обстановка существенно изменилась, а игрок выполнил ряд действий, на которые нужно дать разумный ответ. Конечный автомат должен отслеживать состояние игры и иметь возможность при необходимости выйти из текущего состояния и войти в другое.

Приняв во внимание все изложенное, вы сможете разработать конечный автомат, который хорошо моделирует разумное поведение объекта. Чтобы не быть голословным, рассмотрим некоторые конкретные примеры, начиная с простейшего конечного автомата.

Элементарные конечные автоматы

Вы уже должны были обратить внимание на то, что различные технологии, используемые при создании искусственного интеллекта, порой перекрывают друг друга. В частности, шаблоны представляют собой не что иное, как простейшие конечные автоматы. Сейчас я хочу поближе познакомить вас с конечными автоматами более высокого уровня и их реализацией. Для лучшего понимания рассмотрим объект с простым поведением, которое и смоделируем с помощью конечного автомата.

Большинство игр основаны на некотором конфликте. Является ли конфликт основной идеей игры или всего лишь ее фоном — в любом случае большую часть времени игрок будет искать и уничтожать врагов и/или собирать различные вещи. Нам нужно разработать простейшую модель поведения персонажа игры, которая позволит ему выжить под постоянными атаками игрока. Взгляните на рис. 12.10, на котором показаны взаимосвязи между несколькими состояниями.

Основное состояние 1: Атака.
Основное состояние 2: Отступление.
Основное состояние 3: Случайное движение.
Основное состояние 4: Остановка или временная пауза.
Основное состояние 5: Поиск снаряжения (пищи, энергии, других персонажей и т.п.).
Основное состояние 6: Выбор шаблона и следование ему.

Вы сами должны увидеть разницу между этими состояниями и предыдущими примерами. Перечисленные состояния функционируют на самом верхнем уровне и должны содержать подсостояния или логику для их создания “на лету”. Например, состояния 1 и 2 могут быть реализованы с использованием некоторых детерминированных алгоритмов, в то время как состояния 3 и 4 представляют собой всего лишь несколько строк кода. Состояние же 6 оказывается очень сложным, поскольку, в свою очередь, предусматривает выполнение сложных шаблонов.

Состояние 5 также может представлять собой сложный детерминированный алгоритм (или даже набор таких алгоритмов и заранее созданных шаблонов поиска). Главная идея в том, что мы создаем модель персонажа “сверху вниз”, т.е. сначала разрабатываем искусственный интеллект на верхнем уровне, а затем постепенно спускаемся вниз, реализуя все более низкие уровни.

Возвращаясь к рис. 12.10, вы видите, что выбор основных состояний персонажа осуществляется логикой верхнего уровня, которую можно назвать “волей” или

"программой" персонажа. Имеется множество способов реализации этой логики: случайный выбор, выбор с использованием условных операторов и др. Пока достаточно знать, что состояния верхнего уровня должны выбираться разумным способом, учитывающим текущее состояние игры.

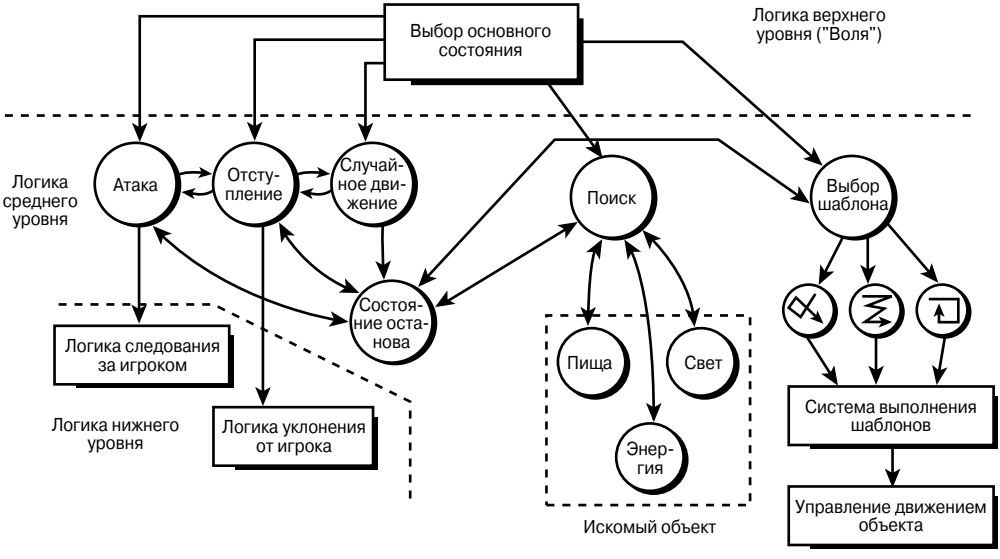


Рис. 12.10. Построение поведенческой модели объекта

Наверняка эти слова кажутся вам слишком общими и ничего не поясняющими. Давайте для лучшего понимания рассмотрим фрагмент кода, реализующий конечный автомат в первом приближении. Данный код нефункционален — это всего лишь набросок, содержащий все важные структурные элементы. Полностью система ИИ такого уровня представляет собой многие страницы кода, которые мы просто не в состоянии разместить в книге. Взяв приведенный код за основу, детализировав и заполнив все пропуски в нем, вы можете воспользоваться им при разработке собственных игр. А теперь приступим к рассмотрению кода, при создании которого предполагалось, что в игре участвуют только игрок и персонаж, управляемый системой ИИ.

```
// Основные состояния
#define STATE_ATTACK  0    // Атака игрока
#define STATE_RETREAT 1    // Отступление
#define STATE_RANDOM  2    // Случайное движение
#define STATE_STOP    3    // Остановка на время
#define STATE_SEARCH  4    // Поиск энергии
#define STATE_PATTERN 5    // Выбор и выполнение шаблона

// Переменные для описания персонажа
int creature_state   = STATE_STOP, // Состояние персонажа
   creature_counter = 0,           // Отслеживание времени
   creature_x      = 320,          // Положение персонажа
   creature_y      = 200,
   creature_dx     = 0,            // Текущая траектория
   creature_dy     = 0;
```

```
// Переменные для описания положения игрока
int player_x = 10,
   player_y = 20;

// Реализация логики персонажа

// Обработка текущего состояния
switch(creature_state)
{
   case STATE_ATTACK:
      {
         // Первый шаг: движение в направлении игрока
         if (player_x > creature_x) creature_x++;
         if (player_x < creature_x) creature_x--;
         if (player_y > creature_y) creature_y++;
         if (player_y < creature_y) creature_y--;

         // Второй шаг: стрельба с вероятностью 20%
         if ((rand() % 5) == 1)
            Fire_Cannon();
      } break;

   case STATE_RETREAT:
      {
         // Движение от игрока
         if (player_x > creature_x) creature_x--;
         if (player_x < creature_x) creature_x++;
         if (player_y > creature_y) creature_y--;
         if (player_y < creature_y) creature_y++;
      } break;

   case STATE_RANDOM:
      {
         // Перемещение персонажа в случайном
         // направлении, которое было определено
         // при входе в это состояние
         creature_x += creature_dx;
         creature_y += creature_dy;
      } break;

   case STATE_STOP:
      {
         // Ничего не делаем
      } break;

   case STATE_SEARCH:
      {
         // Выбираем объект для поиска и движемся по
         // направлению к нему
         if (energy_x > creature_x) creature_x++;
         if (energy_x < creature_x) creature_x--;
         if (energy_y > creature_y) creature_y++;
         if (energy_y < creature_y) creature_y--;
```

```
    } break;

  case STATE_PATTERN:
    {
      // Выполняем выбранный шаблон
      Process_Pattern();
    } break;

  default: break;
} // switch

// Обновляем содержимое счетчика и проверяем, не следует ли
// осуществить выбор нового состояния
if (--creature_counter <= 0)
{
  // Выбираем новое состояние, применяя ту или иную
  // логику. В данном случае используется случайный выбор
  creature_state = rand()%6;

  // В зависимости от выбранного состояния следует
  // выполнить некоторые настройки
  if (creature_state == STATE_RANDOM)
  {
    // Выбор случайного направления
    creature_dx = -4 + rand()%8;
    creature_dy = -4 + rand()%8;
  } // if

  // В случае необходимости - другие настроечные действия

  // Выбор времени для работы в данном состоянии. Исходя
  // из 30 кадров в секунду, выбранное время - от 1 до 5
  // секунд
  creature_counter = 30*(1 + rand()%5);

} // if
```

Давайте обсудим приведенный код. Он начинается с обработки текущего состояния и включает в себя локальную логику, алгоритмы и даже вызовы функций других подсистем ИИ, например для выполнения шаблона. После обработки текущего состояния обновляется содержимое счетчика и, если выполнение текущего состояния завершено, выбирается новое состояние. Если для его работы требуются предварительные настройки, они тут же выполняются. И наконец, выбирается продолжительность пребывания персонажа в новом состоянии.

Этот набросок кода не просто можно, а нужно значительно улучшить. В первую очередь это касается переходов между состояниями, которые могут осуществляться вместе с обработкой текущего состояния, и выбора нового состояния, которое можно и нужно осуществлять на основе информации о текущем состоянии игры, а не вслепую, случайным образом.

Добавление индивидуальности

Индивидуальность, по сути, не что иное, как предсказуемость поведения того или иного персонажа. Например, у меня есть разные знакомые. Среди них есть один малый немалого размера и с горячим характером, и если я (особенно в баре после нескольких

бокалов пива) скажу ему что-то, что ему не понравится, то рискую получить немалый счет от стоматолога, который будет чинить мою челюсть. Другой мой знакомый в такой же ситуации просто промолчит и сделает вид, что ничего не расслышал.

Конечно, люди гораздо более сложны и менее предсказуемы, чем в данном конкретном примере, но для персонажей игры можно обойтись простой моделью распределения вероятностей. Это означает, что у каждого персонажа свои вероятности выбора того или иного поведения в различных ситуациях; набор этих вероятностей, по сути, и определяет их индивидуальность, участвуя в выборе перехода между состояниями. Взгляните на рис. 12.11, чтобы понять, что я хочу этим сказать.

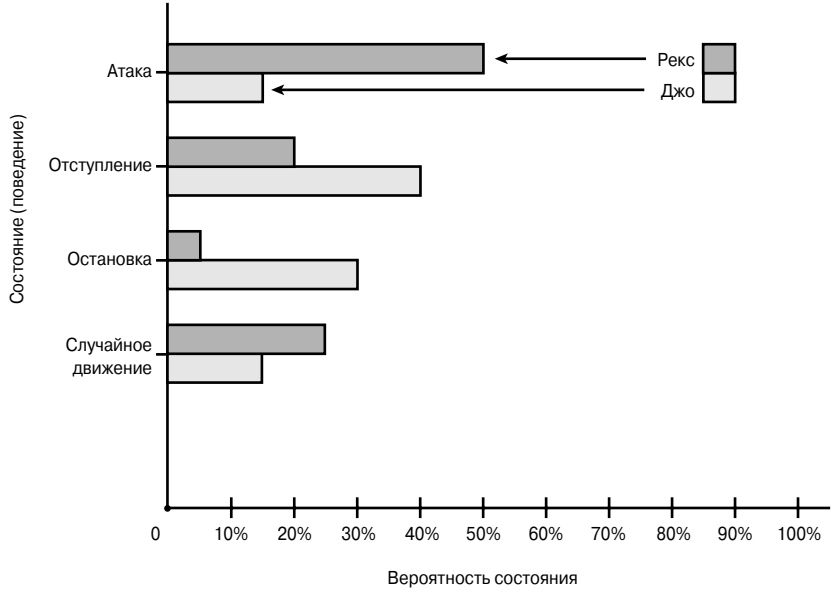


Рис. 12.11. Распределение вероятностей основных состояний разных персонажей игры

Пусть в нашей модели есть четыре основных состояния.

Состояние 1: Атака.
Состояние 2: Отступление.
Состояние 3: Остановка.
Состояние 4: Случайное движение.

Вместо равновероятного выбора нового состояния, как было ранее, мы создаем и используем для каждого персонажа распределение вероятностей, определяющее его индивидуальность. Например, в табл. 12.2 приведены распределения вероятностей для персонажей Рекса и Джо, имитирующих моих друзей, описанных ранее.

Как видите, такая модель не лишена смысла. Если применить приведенное распределение вероятностей, Рекс будет бездумно бросаться в атаку, в то время как Джо предпочтет уклоняться от встречи с противником; кроме того, случайного в его поведении будет меньше, чем у Рекса, от которого в реальной жизни можно ожидать чего угодно.

Разумеется, этот пример полностью придуман, но если вы представите себе персонажи, чье поведение определяется приведенными в табл. 12.2 и на рис. 12.11 распределениями вероятности, то картина будет приблизительно такой, как я ее описал.

Таблица 12.2. Индивидуальные распределения вероятностей

Состояние	*Вероятность для Рекса, %*	*Вероятность для Джо, %*
Атака	50	15
Отступление	20	40
Остановка	5	30
Случайное движение	25	15

Для реализации описанного метода можно, например, воспользоваться таблицей с 20–50 записями (где каждая запись соответствует тому или иному состоянию) и заполнить ее в соответствии с необходимым распределением вероятностей. Например, вот как будут выглядеть 20-элементные таблицы состояний для Рекса и Джо (каждый элемент в такой таблице имеет вес 5%):

```
int rex_pers[20] = {1,1,1,1,1,1,1,1,1,1,2,2,2,2,3,4,4,4,4,4};
int joe_pers[20] = {1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,4,4,4};
```

При разработке систем ИИ к использованию индивидуальности персонажей можно добавить *радиус влияния*. Это означает, что распределение вероятностей персонажа зависит от некоторого фактора, например расстояния до игрока или какого-то другого объекта (рис. 12.12). Как видите, когда персонаж находится достаточно далеко от игрока, он переключается на неагрессивный поиск; если игрок приближается к игроку, тот постарается избежать встречи, а если игрок подойдет слишком близко, попробует атаковать его. Как видите, в каждом диапазоне расстояний используется своя таблица распределения вероятностей.

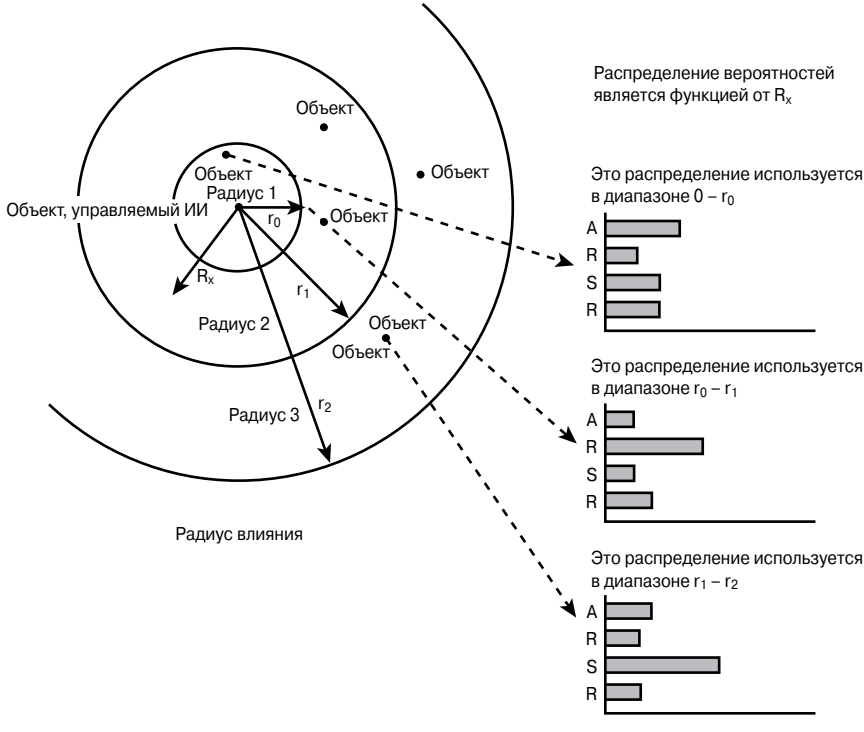


Рис. 12.12. Изменение распределения вероятностей в зависимости от расстояния

Запоминание и обучение

Еще одним важным элементом хорошей системы ИИ является запоминание и обучение. Все рассмотренные технологии ИИ работают только с текущей информацией и никогда не принимают в расчет уже происшедшие события.

Например, если при атаке со стороны персонажа игрок постоянно уходит от атаки вправо, ни одна из рассмотренных технологий ИИ не в состоянии отследить это и предпринять упреждающую атаку с учетом данной особенности игрока.

В качестве другого примера представим, что персонажи игры должны искать боеприпасы точно так же, как и игрок. Однако персонажи вели бы себя гораздо разумнее, если бы запоминали место, где боеприпасы были найдены в последний раз, и начинали поиск именно оттуда, а не искали бы их случайным образом (возможно, с использованием шаблонов).

Это только несколько примеров, демонстрирующих, насколько разумнее выглядели бы персонажи, если бы имели возможность запоминать и обучаться. К счастью, реализовать запоминание несложно, хотя только очень немногие программисты делают это. Обычно им просто не хватает времени (или они полагают, что игра и без этого получилась неплохо). А зря! Запоминание и обучение — очень красивые вещи, и игроки сразу заметят разницу между игрой с использованием этих возможностей и без них. Попытайтесь найти, где в вашей игре можно реализовать простейшее запоминание и обучение искусственного интеллекта без особых сложностей, и посмотрите, насколько при этом изменится игра.

Общая идея, конечно, понятна, и все же я предвижу вопрос: как именно можно реализовать описанную методику? Это зависит от конкретной игры. Рассмотрим, например, рис. 12.13.

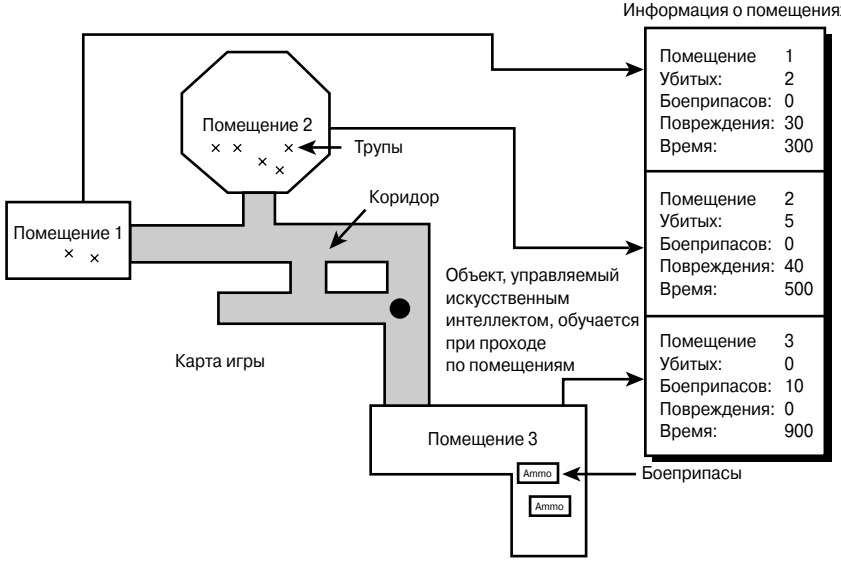


Рис. 12.13. Использование памяти в системе искусственного интеллекта

На нем вы видите карту игрового мира, где каждому помещению соответствует запись, хранящая следующую информацию:

Убитые персонажи
Повреждения, нанесенные игроком
Обнаруженные боеприпасы
Время пребывания в данном помещении

Всякий раз, когда наступает время действия персонажа, в процессе выбора очередного состояния можно обратиться к информации, хранящейся в записи и представляющей собой не что иное, как память персонажа. Например, обнаружив, что в некотором помещении совершенно нет боеприпасов, зато в нем игрок нанес персонажу большой урон, персонаж может тут же попытаться перейти в другое помещение.

Другой пример использования памяти искусственным интеллектом. Если у персонажа заканчиваются боеприпасы, то гораздо разумнее не начинать поиск вслепую, т.е. случайным образом, а просканировать все записи и "вспомнить", в каком помещении остались нетронутые боеприпасы.

Кроме того, персонажи могут обмениваться информацией! Например, если один персонаж встречается с другим, они могут объединить свои записи о помещениях, в которых побывали. Можно реализовать обмен не только информацией, но и, например, оружием, боеприпасами, другим снаряжением. Еще один вариант обмена информацией, когда израненный в бое с игроком персонаж рассказывает другому, свежему и вооруженному до зубов, где именно только что находился игрок, а заодно передает ему часть своего вооружения; сам же он после этого идет восстанавливать силы и боезапас (возможно, используя для этого только что полученную информацию).

Нет предела способам применения запоминания и обучения. Единственное, о чем следует сказать особо: искусственный интеллект должен играть честно. Например, было бы нечестной игрой снабдить ИИ информацией обо всей карте игры. Искусственный интеллект должен быть поставлен в те же условия, и играть по тем же правилам, что и обычный игрок.

СОВЕТ

Многие программисты, работающие над играми, предпочитают использовать для запоминания битовые строки или векторы. Такое представление может оказаться более компактным, а работа с отдельными битами ненамного сложнее, чем с байтами. Кроме того, можно воспользоваться упакованными структурами, облегчающими работу с отдельными битовыми полями.

В качестве примера искусственного интеллекта с запоминанием я разработал простую программу `DEMO12_7.CPP` (ее 16-битовая версия находится в файле `DEMO12_7_16B.CPP`), работа которой показана на рис. 12.14.

Программа представляет собой моделирование поведения искусственных существ, представленных на экране в виде красных муравьев. Все их поведение сводится к простому правилу: случайные перемещения, когда сыт, и поиск еды (показана на экране синим цветом), когда голоден. Когда муравей находит еду, он наедается "до отвала" и продолжает случайные странствия. Когда муравей в очередной раз проголодается, он идет к тому месту, где в последний раз нашел еду. Кроме того, если два муравья сталкиваются друг с другом, они обмениваются информацией о своих путешествиях. Если муравей длительное время не может найти еду, он умирает голодной смертью. В моделируемом мире находится 16 муравьев, но на экране место для детальной информации только для восьми из них. Если у вас мощная машина, вы можете изменить программу, увеличив количество моделируемых особей.

Деревья планирования и принятия решений

До сих пор все технологии ИИ были слишком реакционными и непосредственными, т.е. я хочу подчеркнуть, что они не использовали ни планирования, ни высокоуровневой логики. Поскольку о том, как реализовать низкоуровневый ИИ, вы уже знаете, я хотел

бы поговорить о высокоуровневом искусственном интеллекте, о котором обычно говорят, как о *планировании*.

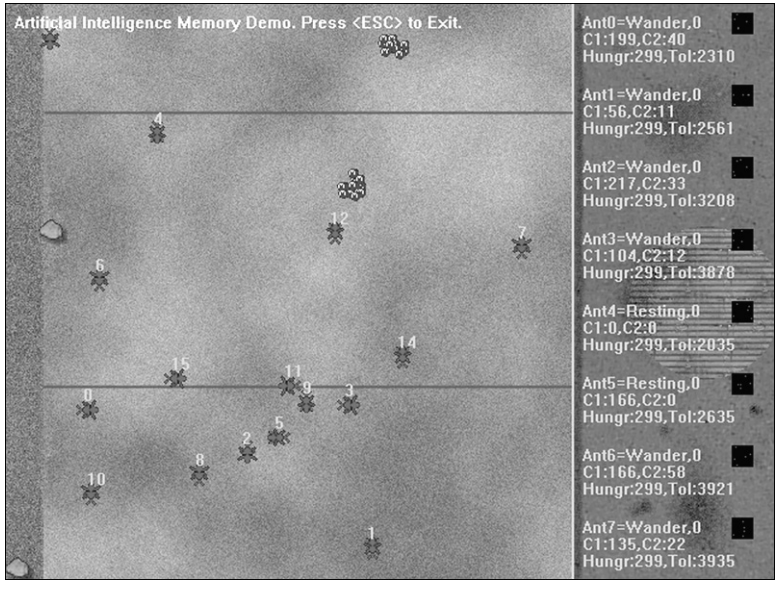


Рис. 12.14. Работа демонстрационной программы DEMO12_7.EXE

План представляет собой просто высокоуровневое множество *действий*, выполняемых в определенном порядке для достижения поставленной *цели*. Кроме действий, в план входит ряд условий, которые должны удовлетворяться перед тем, как будет выполнено то или иное конкретное действие. Например, вот как может выглядеть план похода в театр.

1. Выбрать пьесу, которую бы вам хотелось посмотреть.
2. Отправиться в театр, как минимум, за час до начала спектакля.
3. Оказавшись в театре, приобрести билет.
4. Просмотреть пьесу и отправиться домой.

Выглядит более-менее разумно. Однако я опустил множество существенных деталей. Например, в какой именно театр нужно ехать? Хватит ли одного часа, чтобы добраться до театра? Что делать, если не хватает денег на билет? Ну и так далее... Эти детали могут быть важны (или не важны), в зависимости от того, насколько сложный план вы строите. Однако вообще говоря, должно быть столько условий и подпланов, чтобы при выполнении основного плана не возникало ни одного вопроса.

Реализация алгоритмов планирования для искусственного интеллекта игровой программы основывается на той же концепции, что и ранее. У нас есть объект, управляемый системой ИИ, и мы хотим, чтобы он следовал некоторому плану для достижения некоторой цели. Следовательно, необходимо смоделировать этот план с помощью некоторого языка программирования; обычно это C/C++, но можно использовать и некоторый специализированный высокоуровневый язык сценариев. В любом случае, кроме моделирования плана, вы должны смоделировать все объекты, являющиеся частью плана: действия, цели и условия. Каждый из перечисленных элементов может быть представлен простой структурой или классом C/C++. Например, цель может выглядеть следующим образом:

```
typedef struct GOAL_TYP
{
  int  class;          // Класс цели
  char* name;          // Название цели
  int  time;           // Время, отпущенное на достижение цели
  int*  subgoals;      // Указатель на список подцелей,
                       // которые должны быть достигнуты
  int (*eval)(void); // Указатель на функцию, определяющую,
                       // достигнута ли цель

  // Прочие данные

} GOAL, *GOAL_PTR;
```

Конечно, это определение — всего лишь пример, и в вашем конкретном случае полей может оказаться намного больше; главное, чтобы вы поняли идею. Вы создаете структуру, которая может представлять любую цель в вашей игре — от "взорвать мост" до "найти еду".

Следующая требующаяся вам структура — это структура действия, которая представляет нечто, что объект должен сделать в качестве части плана по достижению цели. Выбор конкретной структуры — за вами; опять же, данная структура должна позволять описать все, что должен уметь делать объект. Вот пример подобной структуры:

```
typedef struct ACTION_TYP
{
  int  class;              // Класс действия
  int* name;               // Название действия
  int  time;               // Время, выделенное для
                           // выполнения действия
  RESOURCE*  resource; // Указатель на запись,
                           // описывающую ресурсы,
                           // которые могут понадобиться
                           // при выполнении действия
  CONDITIONS* cond;    // Указатель на запись,
                           // описывающую все условия,
                           // которые должны выполняться до
                           // того, как данное действие
                           // будет выполнено
  UPDATES*   update;   // Указатель на запись,
                           // описывающую все обновления и
                           // изменения, которые должны быть
                           // выполнены по завершении действия
  int (*action_functions)(void);
                           // Указатель на функцию(и), которая
                           // выполняет данное действие
} ACTION, *ACTION_PTR;
```

Понятно, что это весьма абстрактный пример. Очевидно, что в вашей реализации эта структура может быть совершенно другой.

Кодирование планов

Существует ряд способов кодирования планов. Ваш код, реализующий действия, цели и план, может быть непосредственно разработан на языке C/C++ и являться неотъемлемой частью программы. Такая технология "прошитого", или жестко закодированного (hard coded) в программе плана была широко распространена в прошлом.

Более элегантный метод кодирования плана состоит в использовании *правил продукции*, или просто *продукций*[1], и деревьев принятия решений. Продукция представляет собой логическое утверждение с рядом *посылок* и *следствием*:

```
IF X OP Y THEN Z
```

Здесь X и Y — посылки, Z — следствие, а OP может быть любой логической операцией, например AND, OR и т.д. Кроме того, X и Y могут быть составлены из других продукций, т.е. продукции могут быть вложенными. Рассмотрим, например, следующий оператор:

```
if (P > 20) AND (damage < 100) THEN consequence
```

Или на языке C/C++:

```
if (power > 20) && (damage < 100)
{
  consequence();
} // if
```

Таким образом, фактически продукция представляет собой условное выражение, а прошитый план является не чем иным, как набором условных операторов вместе с действиями и целями. Цель разработки "планировщика" состоит в том, чтобы смоделировать поведение объектов немного абстрактнее. Хотя никто не мешает вам закодировать продукции непосредственно, лучше все же создать структуру, которая может читать продукции, содержать действия и цели и представлять план.

Одной из структур, способной помочь в реализации этой системы, является дерево принятия решений. Как показано на рис. 12.15, это простое бинарное дерево, каждый узел которого представляет продукцию и/или действие.

Однако вместо использования жесткого кодирования для реализации дерева мы загружаем его из файла или строим из данных, предоставленных программистом или разработчиком уровня. Этот способ общего назначения, в отличие от жесткого кодирования, не требует перекомпиляции программы при изменении дерева принятия решения. Давайте в качестве примера рассмотрим небольшой язык планирования для управления боевым роботом с использованием некоторых входных переменных и набора действий.

Приведем параметры, которые могут быть протестированы системой ИИ.

DISPLY	Расстояние до игрока (0 – 100)
FUEL	Остаток горючего (0 – 100)
AMO	Остаток боеприпасов (0 – 100)
DAM	Текущие повреждения (0 – 100)
TMR	Текущее время игры в виртуальных минутах
PLAYST	Состояние игрока (атакующее, не атакующее)

Действия, которые может выполнять робот.

FW	Огонь по игроку
SD	Самоуничтожение
SEARCH	Поиск игрока
EVADE	Уклонение от игрока

[1] Данное название взято из теории языков программирования (см., например, А. Ахо, Р. Сети, Д. Ульман. *Компиляторы: принципы, технологии и инструменты* — М. : Издательский дом "Вильямс", 2001). Это не случайное совпадение, так как один из методов кодирования искусственного интеллекта — разработка собственного языка программирования для описания целей, планов и действий объектов и использование в игре соответствующего транслятора. Преимущества такого подхода, несмотря на его сложность, очевидны. Впрочем, об этом будет сказано немного позже. — *Прим. ред.*

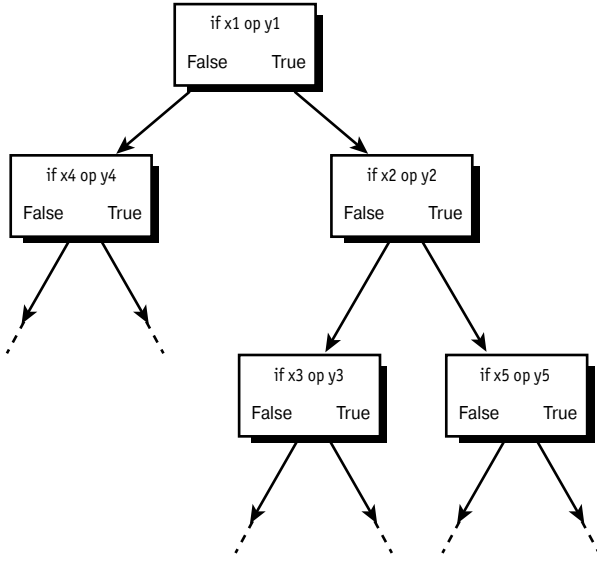


Рис. 12.15. Дерево принятия решений

Теперь разработаем структуру дерева принятия решений. Будем считать, что в каждом узле может быть проверена одна посылка (или две, связанные логическим оператором AND или OR), к которой может быть применен оператор логического отрицания NOT. Посылки сами по себе представляют сравнения входных переменных и констант при помощи операторов >, <, == и !=. Кроме того, каждый узел имеет две ветви — TRUE и FALSE, а также список возможных действий (до восьми пунктов) (см. рис. 12.15). Ниже приведена структура, которая может быть использована для реализации узла.

```
typedef struct DECNODE_TYP
{
  int operand1, operand2;   // Первая пара операторов
  int comp1;                // Оператор сравнения

  int operator;             // Оператор объединения

  int operand3, operand4;   // Вторая пара операторов
  int comp2;                // Оператор сравнения

  ACTION* act_true;         // Список действий при выпол-
  ACTION* act_false;        // нении и невыполнении условия

  DECNODE_PTR* dec_true;  // Ветви при выполнении и
  DECNODE_PTR* dec_false; // невыполнении условия

} DECNODE, *DECNODE_PTR;
```

Как видите, эта структура может реализовать различные типы посылок, состоящих из одного или двух сравнений, причем сравниваться могут как две переменные, так и переменные с константами.

Программная реализация данной технологии не представляет особого труда, так что я не буду подробно на ней останавливаться. Просто скажу, что вы должны следовать по уз-

лам, вычислять условные выражения в них и, в зависимости от полученного результата, выполнять те или иные действия и переходы к другим узлам дерева принятия решений.

После этого вам нужно лишь разработать само дерево принятия решений, которое и будет определять поведение робота в разных ситуациях. Рассмотрим, например, систему управления огнем. Понятно, что приводимый ниже план далеко не полон, но в качестве примера плана, определяющего тактику ведения огня, его можно принять. Грубо его можно сформулировать на русском языке следующим образом.

- ***Если*** игрок близко ***и*** повреждения малы, ***то*** атаковать игрока.
- ***Если*** игрок далеко ***и*** горючего много, ***то*** искать игрока.
- ***Если*** повреждения велики ***и*** игрок близко, ***то*** убегать от игрока.
- ***Если*** повреждения велики, ***и*** боеприпасов нет, ***и*** игрок близко, ***то*** самоуничтожиться.

Вот таков мой маленький псевдоплан действий робота. Понятно, что полный план может содержать десятки или даже сотни таких предложений. Но самое приятное в нем, что дизайнер игры не должен его кодировать, а может использовать вместо этого графический инструментарий для построения схемы наподобие показанной на рис. 12.16, которую затем легко преобразовать в программу с использованием разработанного вами языка планирования.

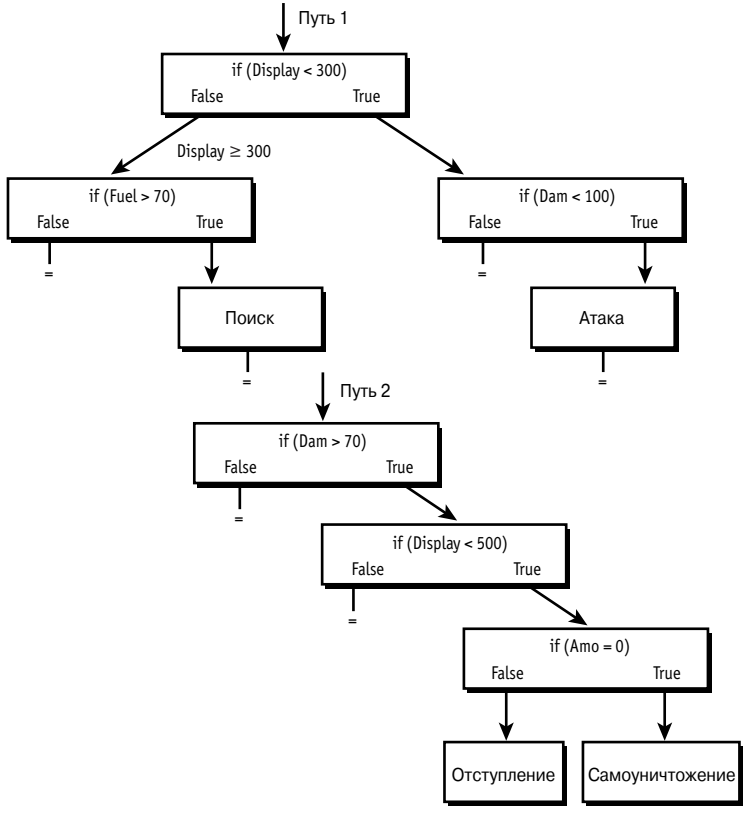


Рис. 12.16. Дерево принятия решений

Итак, главное в использовании описанного метода — это разделение представления плана и его обработки. Вы создаете обработчик плана, который двигается по ветвям дерева принятия решений, вычисляя условные выражения и выполняя необходимые действия. Завершим же мы наше рассмотрение формальным алгоритмом планирования, который не только достигает поставленной цели, но и выполняет анализ плана.

Реализация реальных планировщиков

Вы познакомились с тем, как реализовать часть планирования, отвечающую за вычисление условных выражений, и часть, выполняющую действия. Цель плана представляет собой просто формализацию того факта, что каждый план должен иметь цель, которая достигается в процессе его выполнения. Соответственно, когда выполнение плана завершается, в обязательном порядке требуется проверка того, достигнута ли поставленная цель. Возможна также ситуация, когда наряду с основным планом имеются параллельно выполняющиеся подпланы, и тогда для достижения цели основным планом требуется достижение целей всеми подпланами.

Непонятно? Рассмотрим глобальный план "Все роботы встречаются в точке (x,y,z)". Эта цель не может быть достигнута, пока не будет достигнута цель "Переместиться в точку (x,y,z)" для каждого робота. Кроме тог, если один из роботов не в состоянии достичь заданной точки, планировщик должен обнаружить эту ситуацию и отреагировать на нее. В этом и состоит идея контроля и анализа плана. Чуть позже я еще вернусь к этой теме, а пока рассмотрим, каким образом может быть представлен план.

Сам по себе план может быть неявно представлен в дереве решений либо представлять собой список решений и действий, каждое из которых, в свою очередь, является деревом или последовательностью. Какой способ представления выбрать — решать вам. Важно лишь, что вы должны быть способны сформулировать план, последовательность действий по его выполнению и цель. Действия обычно включают в себя вычисление условных выражений и выполнение поддействий на низком уровне, например таких, как перемещение из одной точки в другую или стрельба. Термин *действие* на высших уровнях логики искусственного интеллекта означает выполнение задачи типа "Уничтожить игрока" или "Захватить укрепление"; действия же на нижнем уровне выполняются непосредственно процессором игры.

Итак, в предположении, что план представляет собой некий массив или связанный список, который можно обойти, планировщик может иметь следующий вид:

```
while( План не пуст, и цель не достигнута )
{
  Получить очередное действие плана
  Выполнить действие
} // while
```

Конечно, вы должны понимать, что это всего лишь абстрактная реализация планировщика. В реальной программе вся эта работа должна выполняться параллельно с решением других задач. Понятно, что вы не можете постоянно находиться в цикле while в ожидании, пока не будет достигнута цель плана. Планировщик следует реализовать в виде конечного автомата или подобной структуры и отслеживать состояние плана во время работы игры.

Данный алгоритм планирования несколько "туповат". Он не принимает во внимание, что в будущем действия могут оказаться невозможны, а план — бесполезен. В результате планировщик должен проводить дополнительный анализ плана и убеждаться, что он имеет смысл. Например, если план состоит во взрыве моста, но в ходе его выполнения мост оказывается взорван кем-то другим, планировщик должен суметь сделать вывод о

невозможности выполнения плана и прекратить его выполнение. Это можно сделать, рассматривая цели плана и проверяя, не достигнуты ли эти цели другими персонажами, и если да, то выполнение такого плана следует прекратить.

Планировщик должен также рассматривать события или состояния, которые могут сделать план невозможным. Например, в некоторый момент при выполнении плана может потребоваться синий ключ, который был потерян. Такая ситуация также должна отслеживаться планировщиком, который просматривает план в контексте того, что может понадобиться в будущем, и который должен уметь корректировать план с учетом сделанных выводов. Согласитесь, что при выполнении плана "Пройти 1000 километров и взорвать форт" глупо будет пройти эту 1000 километров и выяснить, что взрывчатки-то у тебя нет! Это уже не искусственный интеллект, а искусственная глупость. Планировщик должен рассмотреть цель, отследить все необходимое для ее выполнения и убедиться, что персонаж, следующий плану, имеет необходимое количество взрывчатки или найдет ее по дороге.

Согласитесь, глупо постоянно отказывать себе в удовольствии взорвать вражеский форт только потому, что у тебя нет взрывчатки, которой можно разжиться в пути. Эта мысль приводит нас к классификации требований плана в соответствии с их *приоритетами*. Например, если для выполнения какого-то плана мне необходимо лазерное ружье, а оно только одно на всю игру да и то давно разбито, то выполнять такой план не следует. С другой стороны, если для подкупа президента мне требуется 1000 золотых монет, а у меня их всего три, то не стоит сразу отказываться от плана, особенно если до резиденции президента далеко, — ведь по дороге можно найти недостающую сумму. Следовательно, отбрасывать такой план нет оснований.

И наконец, когда план находится в стадии выполнения, не обязательно полностью прекращать его выполнение. Можно изменить его или выбрать новый план на основе уже сделанного. Например, можно изначально иметь, скажем, три плана действий — один основной и два резервных на случай, если выполнение основного плана окажется невозможным.

Планирование — очень мощный инструмент искусственного интеллекта, применимый, по сути, в любой игре. Даже если вы пишете клон Quake, в котором не нужно особенно размышлять, а нужно только стрелять, вам все равно потребуется глобальный планировщик, руководящий персонажами, главную цель которых можно сформулировать как "Стой здесь и убей игрока". Если же вы работаете над военным симулятором в стиле Command and Conquer, то планирование — единственный способ написать такую игру.

Наилучший способ использования планирования в реальной разработке игр — это разработать собственный язык планирования и дать его дизайнеру для составления плана вместе с набором переменных и объектов, которые могут быть его частью. Это позволит дизайнеру разработать такие далеко идущие планы, которые вы не смогли бы не только жестко закодировать, но даже и просто придумать!

Поиск пути

Говоря попросту, *поиск пути* (pathfinding) представляет собой вычисление и выполнение движения по пути из точки *p1* к целевой точке *p2* (рис. 12.17). При отсутствии препятствий простейший способ попасть в целевую точку — двигаться по прямой в ее направлении, пока целевая точка не будет достигнута. Однако задача существенно усложняется, если на пути встречаются препятствия, которые придется огибать.

Метод проб и ошибок

Когда на пути появляются небольшие выпуклые препятствия, обычно используется алгоритм, состоящий в том, чтобы при столкновении с препятствием отойти назад, вы-

полнить поворот вправо или влево на 45–90° и переместиться на некоторое предопределенное расстояние (AVOIDANCE_DISTANCE). После этого искусственный интеллект вновь определяет направление на целевую точку и повторяет попытку добраться к ней по прямой. Пример работы этого алгоритма показан на рис. 12.18.

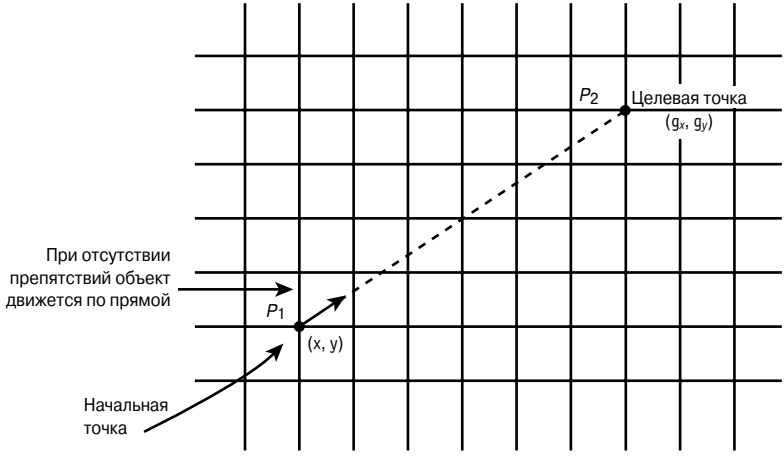


Рис. 12.17. Поиск пути от точки к точке

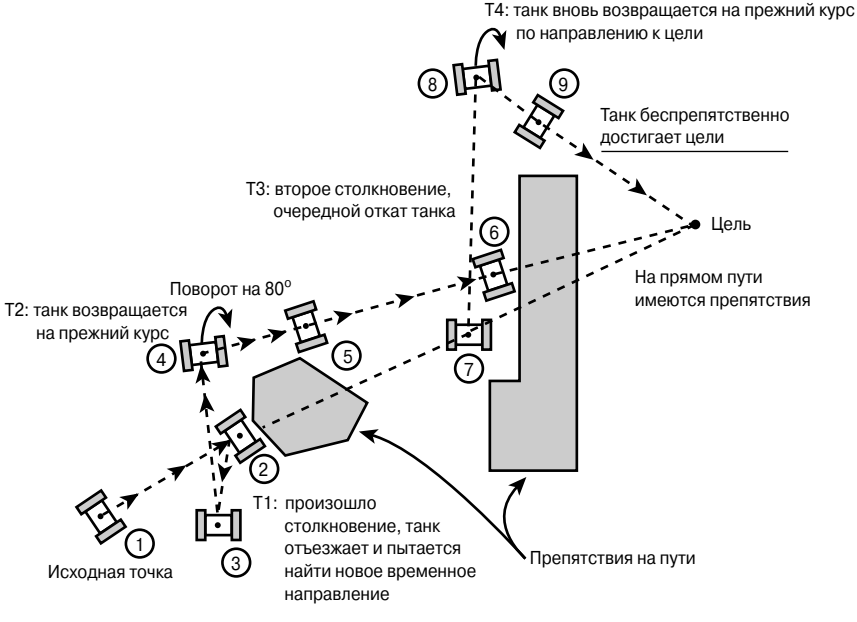


Рис. 12.18. Метод проб и ошибок

Хотя данный алгоритм не настолько надежен и “умен”, как хотелось бы, тем не менее он работает благодаря использованию принципа случайности. Каждый раз при новой попытке объект поворачивается в случайном направлении, так что рано или поздно путь вокруг препятствия будет найден.

Обход по контуру

Другой метод обхода препятствий состоит в обходе по контуру. Данный алгоритм отслеживает контур препятствия на дороге объекта. Реализация данного алгоритма состоит в перемещении объекта вокруг препятствия и периодической проверке, продолжает ли отрезок между объектом и целью пересекать препятствие. Если нет — объект может двигаться по направлению к своей цели; в противном случае обход препятствия продолжается. Пример применения такого алгоритма показан на рис. 12.19.

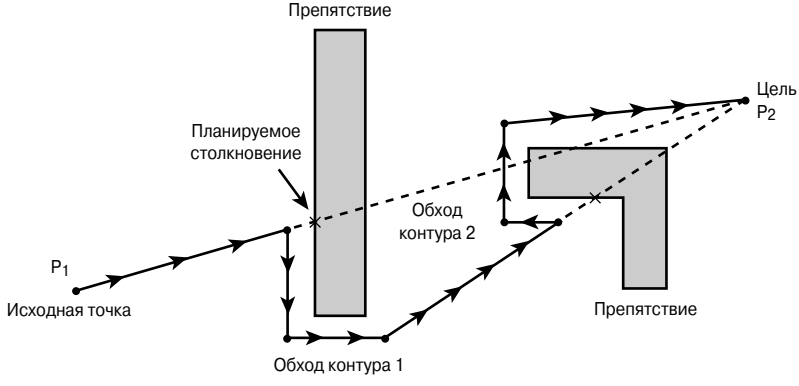


Рис. 12.19. Обход по контуру

Этот алгоритм вполне работоспособен, хотя и выглядит не самым умным, поскольку, как правило, объект движется не по очевидному с первого взгляда кратчайшему пути. Вы можете скомбинировать два описанных метода поиска пути в один: сначала объект движется методом проб и ошибок, и если не достигает цели за заранее заданное время, то в действие вступает алгоритм обхода по контуру.

Конечно, в играх, где взгляду игрока недоступно все поле сразу, заметить примитивность метода можно лишь по тому, что достижение цели занимает несколько больше времени, чем минимально необходимое. Однако в стратегических играх, где взгляду игрока доступно поле целиком, перемещение по контуру выглядит достаточно жалко. Давайте посмотрим, нельзя ли воспользоваться каким-то более "умным" алгоритмом?

Избежание столкновений

При использовании данного метода создаются виртуальные пути вокруг объектов, состоящие из серии точек или векторов, которые позволяют обойти объект по вполне разумному пути. Этот путь может быть вычислен с использованием алгоритма кратчайшего пути (о нем речь идет чуть позже) либо создан вручную вами или дизайнером игры с помощью соответствующего инструментария.

Вокруг каждого большого препятствия создается невидимый для всех, кроме управляемого искусственным интеллектом персонажа, путь. Когда объекту требуется обойти препятствие, он запрашивает кратчайший путь обхода для данного препятствия и получает его. Тем самым гарантируется, что персонаж всегда знает, как обойти то или иное препятствие. Конечно, для того чтобы внести в игру разнообразие, можно разработать для препятствия несколько различных путей обхода или добавить небольшой "шум" к пути с тем, чтобы объект не следовал всегда в точности по одному и тому же пути. Проиллюстрирован описанный метод на рис. 12.20.

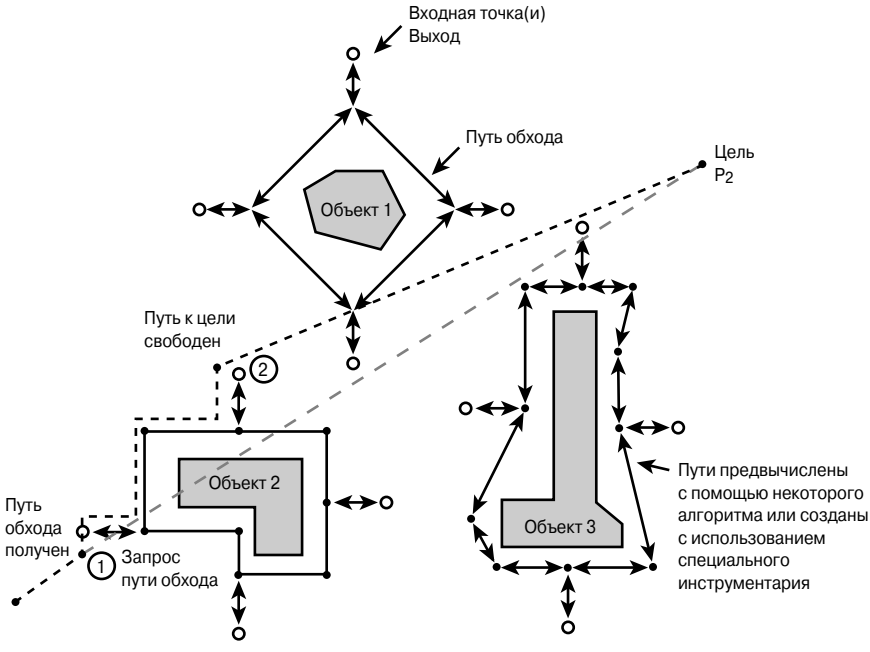


Рис. 12.20. Использование предварительно созданных путей обхода препятствий

Описанный метод подводит к другой идее: почему бы не иметь предвычисленные пути для всех важных точек игры? Тогда, если объект должен добраться от точки p_i в точку p_j, вместо определения направления и обхода препятствий можно воспользоваться предвычисленным путем.

Поиск путей с использованием промежуточных пунктов

Пусть в нашей игре создан действительно сложный мир с препятствиями разнообразных типов. Конечно, можно создать персонаж, достаточно разумный для того, чтобы перемещаться по такому миру, опираясь только на свои силы; но так ли уж это необходимо? Можно просто создать сеть путей, которая соединяет все важные точки игры. Каждый узел такой сети является промежуточным пунктом (waypoint), а дуги такой сети представляют собой направление вектора движения и расстояние от одного узла сети до следующего.

Например, пусть у нас есть план движения и нам нужно переместить персонаж из его текущего положения по мосту и привести в город. Вообще-то это сложная задача, но при наличии сети путей она сводится к поиску такого пути в город, который проходит по мосту, после чего нужно просто следовать по этому пути. В таком случае персонаж гарантированно придет куда требуется, благополучно избежав при этом столкновений с препятствиями. На рис. 12.21 показан пример вида такого мира сверху, вместе с сетью путей. Отмеченный путь и есть тот, который мы искали. Запомните, что такая сеть не только обходит все препятствия, но и должна содержать пути ко всем сколь-нибудь важным точкам игры (например, к подпространственным переходам в вашей вселенной).

В этой большой схеме есть два сложных момента: во-первых, следовать по пути и, во-вторых, создать структуру данных, которая могла бы представлять эту сеть. Начнем со следования по пути.

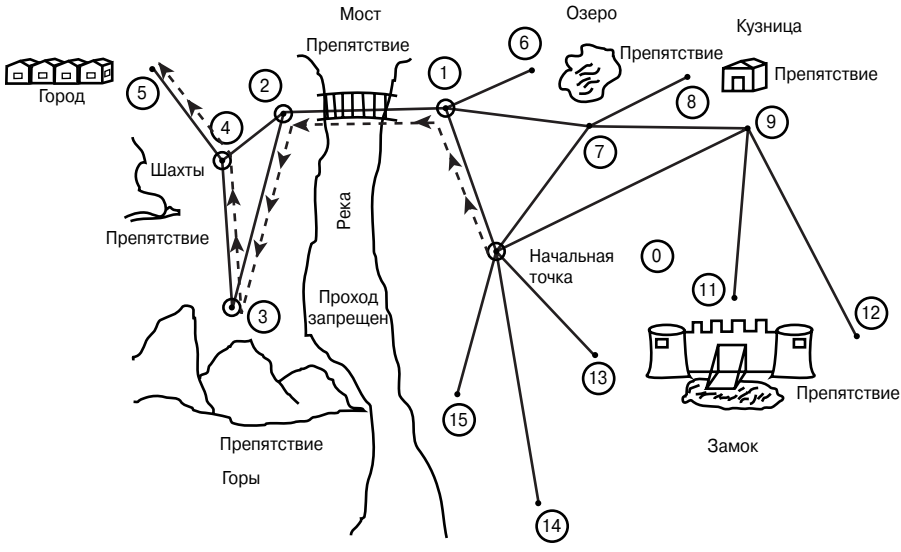


Рис. 12.21. Сеть путей на карте мира

Предположим, у нас есть путь от точки p_1 к точке p_2, состоящий из n узлов, представленных следующей структурой:

```
typedef struct WAYPOINT_TYP
{
  int   id;          // Идентификатор промежуточной точки
  char* name;        // Имя промежуточной точки
  int   x, y;        // Положение промежуточной точки
  int   distance;    // Расстояние до следующей точки
  WAYPOUNT_TYP*
      next;          // следующая точка в списке
} WAYPOINT;
```

Это всего лишь пример; в своей программе вы можете использовать совершенно иную структуру данных. Теперь предположим, что есть шесть промежуточных точек, включая начальную p_1 и конечную p_2, как показано на рис. 12.21.

```
WAYPOINT path[6] = {
  {0, "START",  x0, y0, d0, &path[1]},
  {1, "ONPATH", x1, y1, d1, &path[2]},
  {2, "ONPATH", x2, y2, d2, &path[3]},
  {3, "ONPATH", x3, y3, d3, &path[4]},
  {4, "ONPATH", x4, y4, d4, &path[5]},
  {5, "ONPATH", x5, y5, d5, NULL   }};
```

Прежде всего следует обратить внимание на то, что, несмотря на статическое выделение массива для хранения пути, его точки связаны одна с другой. Последний указатель в этом связанном списке равен NULL, поскольку это поле принадлежит конечной точке нашего пути.

Для того чтобы следовать по указанному пути, необходимо принять во внимание несколько вещей. Например, вы должны добраться до первого узла пути или узла, следующего за ним, что может создать определенную проблему. Решить ее можно, найдя ближайшую к текущему положению объекта входную точку или точку на интересующем вас

пути и двигаясь к ней с использованием одного из описанных ранее алгоритмов. После того как вы достигнете точки в сети, дальнейшее перемещение по пути представляет собой простую задачу.

Следование по пути

Путь представляет собой серию точек, между которыми гарантированно нет никаких препятствий. Почему? Да потому, что вы сами создавали эти пути и точно знаете это! В результате вы можете просто перемещать объект из одной точки, описываемой структурой WAYPOINT, в следующую, и продолжать поступать таким образом до тех пор, пока не доберетесь до последней, целевой точки.

```
Найти ближайшую точку WAYPOINT на интересующем нас пути.

while( цель не достигнута )
{
  Вычислить траекторию от данной точки к следующей
  и переместиться по ней

  По достижении следующей точки траектории обновить
  текущую и очередную точки

} // while
```

Таким образом, вы просто следуете по ряду точек до тех пор, пока не достигаете целевой точки. Вектор перемещения от одной точки WAYPOINT до следующей находится элементарно:

```
// Начинаем со стартовой точки
WAYPOUINT_PTR current = &path[0];

// Находим вектор в направлении следующей точки:
trajectory_x = current->next.x - current->x;
trajectory_y = current->next.y - current->y;

// Нормализуем полученный вектор
normalize(&trajectory_x, &trajectory_y);
```

Процесс нормализации делает длину вектора равной 1.0, оставляя его направление неизменным. Это достигается путем деления каждой компоненты вектора на его длину. Как только объект достигает очередной точки на пути, она становится текущей и происходит вычисление направления движения в точку, следующую за ней. Я не расписываю здесь детально, как определить, что очередная точка достигнута объектом. Для этого можно просто вычислять расстояние от объекта до точки и считать точку достигнутой, когда оно становится меньше некоторого заранее заданного.

Существуют проблемы и с поиском пути. Найти путь — задача не менее сложная, чем добраться от точки к точке при наличии препятствий. Здесь все решается в основном на уровне используемых структур данных. Дело в том, что одни и те же отрезки между точками могут использоваться в различных путях (соответствующий пример показан на рис. 12.22) и, будут ли одни и те же промежуточные точки повторно использоваться в разных путях, зависит от конкретной реализации алгоритма.

Гонки

Хорошим примером использования путей являются игры с гонками. Представьте, что на треке должно находиться несколько автомобилей, которые должны двигаться по треку, при

этом избегая столкновений с игроком и ведя себя достаточно разумно, с точки зрения постороннего наблюдателя. Для таких игр технология путей подходит особенно хорошо.

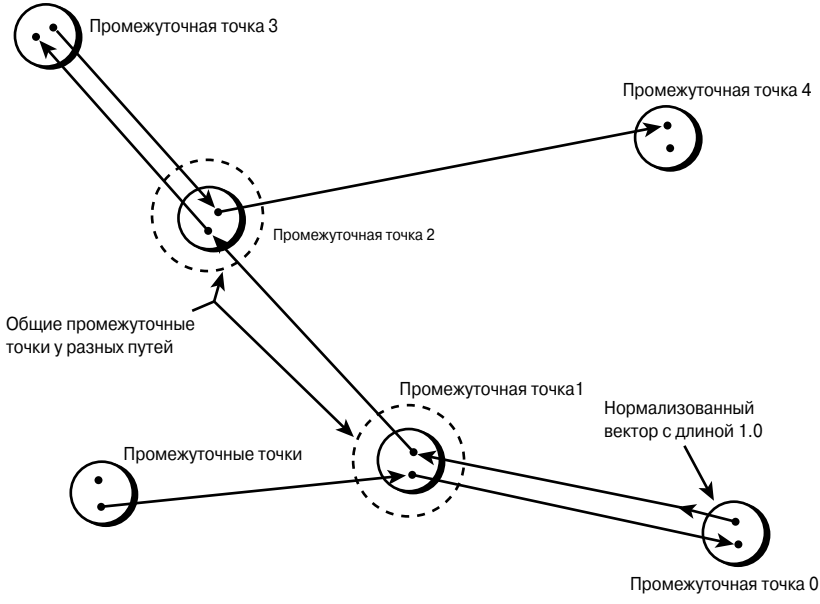


Рис. 12.22. Пути с общими промежуточными точками

Вы можете создать, например, полтора десятка различных путей, у каждого из которых имеются свои особенности. Каждый автомобиль, управляемый системой ИИ, движется в процессе игры по своему пути.

В случае аварии, поломки и прочего автомобиль выбирает ближайший путь и следует по нему. Возможен также переход от одного пути к другому в процессе движения. Все это делает наблюдаемую картину достаточно реалистичной.

На прилагаемом компакт-диске находится демонстрационная программа `DEMO12_8.CPP` (и ее 16-битовая версия `DEMO12_8_16B.CPP`), представляющая собой гонку одного автомобиля по единственному пути. Ознакомьтесь с исходным текстом демонстрационной программы, поработайте с ним. Однако не забывайте о том, что это приложение DirectX, так что при его компиляции вы должны подключить соответствующие библиотеки.

Надежный поиск пути

Теперь я бы хотел поговорить о *реальном* поиске путей, другими словами, о применении вычислительных алгоритмов для поиска пути между точками p_1 и p_2. Для решения данной задачи имеется ряд алгоритмов, однако все они не применимы для работы в реальном времени и потому не могут использоваться в играх. Однако они вполне пригодны для построения путей в соответствующих инструментах, а также в играх, но с определенными упрощениями.

Все алгоритмы такого типа работают с графообразными структурами, представляющими игровое пространство в виде множества узлов с дугами, указывающими, какие именно узлы могут быть достигнуты из данного. Обычно с каждой дугой связана ее цена. Типичный граф такого вида показан на рис. 12.23.

Поскольку мы имеем дело с двух- и трехмерными играми, вы можете решить, что граф представляет собой наложенную на игровое поле решетку, в которой каждая ячейка

неявно связана со всеми восемью соседними ячейками; при этом цена представляет собой не что иное, как расстояние между ячейками (рис. 12.24).

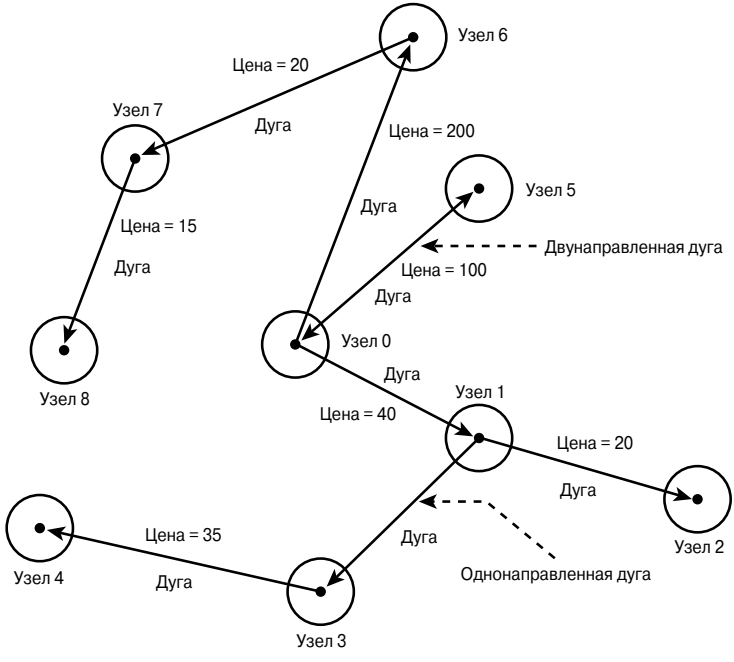


Рис. 12.23. Граф сети

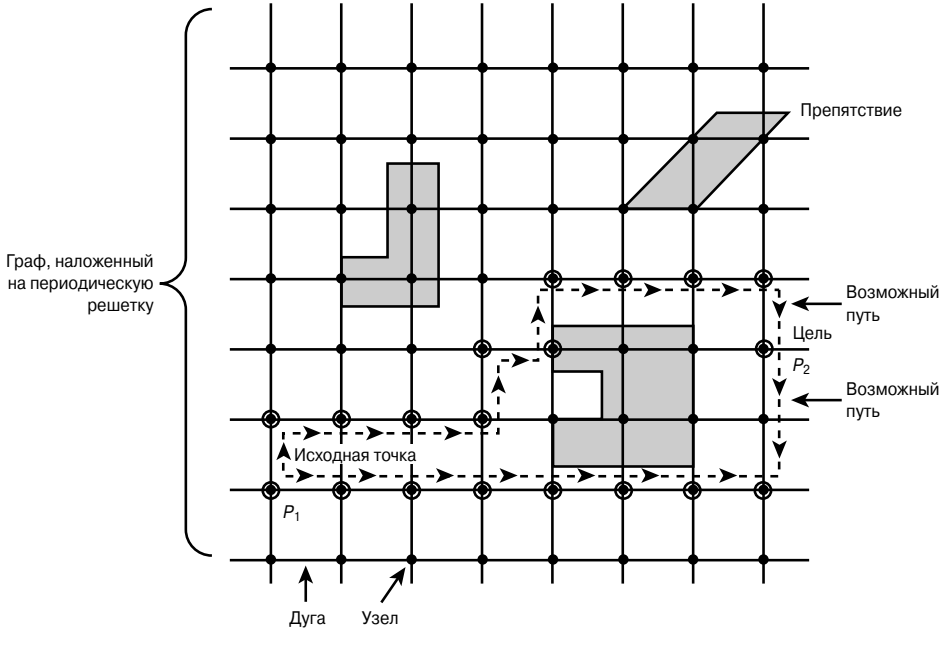


Рис. 12.24. Создание графа с использованием решетки на игровом поле

В любом случае, каким бы ни было представление игрового поля в виде графа, наша задача — найти наиболее короткий путь между точками p_1 и p_2, на котором не встретится ни одного препятствия (поскольку понятие препятствия в графе отсутствует, их попросту не может оказаться на найденном пути). Существует ряд алгоритмов, решающих поставленную задачу, с некоторыми из них мы познакомимся немного подробнее (хотя и не настолько, насколько они того заслуживают).

Поиск в ширину

Этот поиск одновременно охватывает все направления, посещая прежде всего все узлы, доступные из нашей стартовой точки, затем все узлы, доступные из уже посещенных, и т.д. Это достаточно простой и неэффективный алгоритм, поскольку он никак не учитывает интересующее нас направление на цель. Вот как выглядит псевдокод этого алгоритма (схема его работы представлена на рис. 12.25).

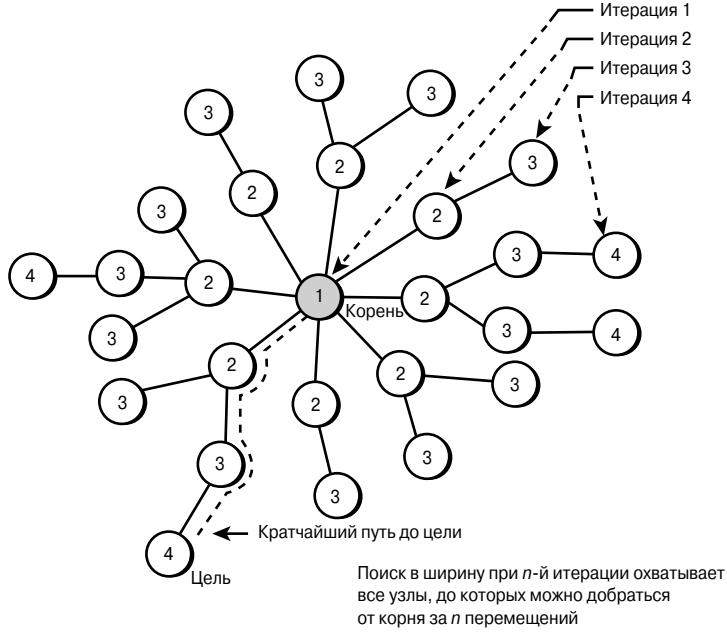


Рис. 12.25. Поиск пути в ширину

```
void BreadthFirstSearch(NODE r)
{
  NODE s;  // Используется для сканирования
  QUEUE q; // Очередь

  // Очередь пуста
  EmptyQ(q);

  // Посещаем узел
  Visit(r);
  Mark(r);

  // Вносим узел r в очередь
```

```
  InsertQ(r,q);

  // Цикл, пока очередь не пуста
  while( q не пуста )
  {
    NODE n = RemoveQ(q);

    for( Все непомеченные узлы s, смежные с n)
    {
      // Посещаем узел
      Visit(s);
      Mark(s);

      // Вносим узел s в очередь
      InsertQ(s,q);
    } // for
  } // while
} // BreadthFirstSearch
```

Двунаправленный поиск в ширину

Этот алгоритм аналогичен только что рассмотренному с тем отличием, что одновременно выполняются два поиска: один из исходной точки, а второй со стороны цели. Когда они перекрываются, вычисляется кратчайший путь (рис. 12.26).

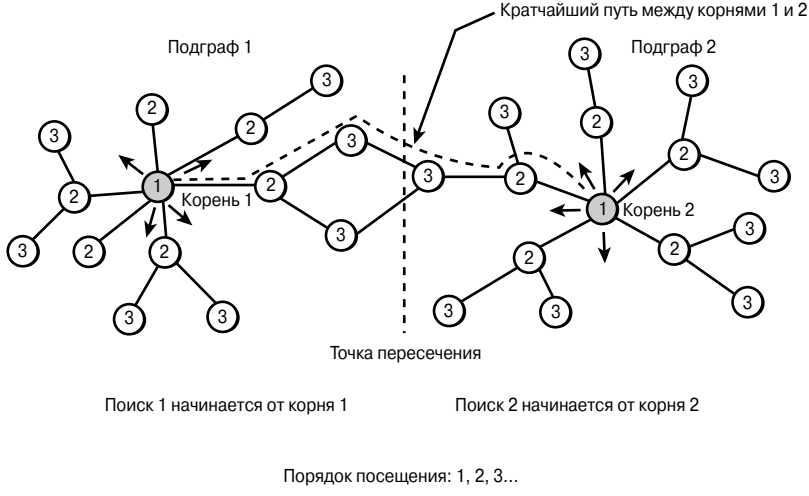
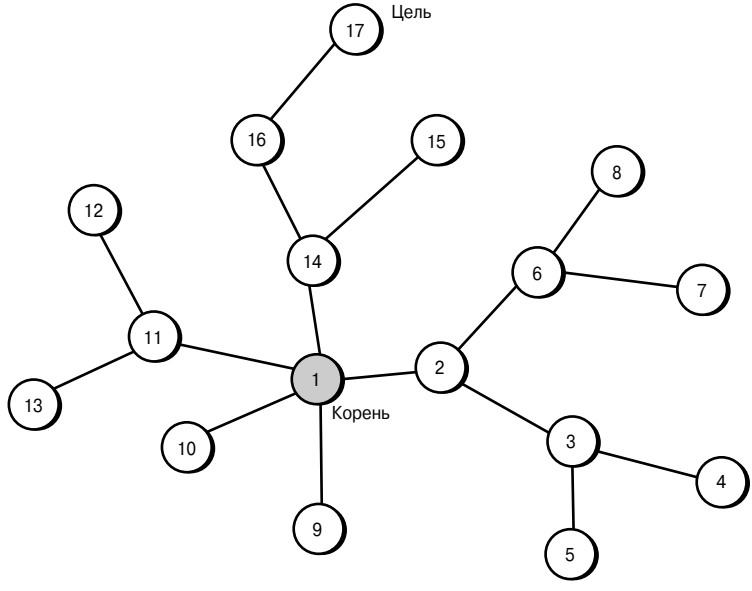


Рис. 12.26. Двунаправленный поиск в ширину

Поиск в глубину

Этот вид поиска противоположен поиску в ширину. Поиск в глубину просматривает одно направление до тех пор, пока его узлы не исчерпаются или пока не будет достигнута цель. Затем выполняется поиск в другом направлении и т.д. Проблема заключается в том, что при этом поиске требуется некоторое останавливающее поиск правило, что-то вроде "Если цель находится на расстоянии в 100 единиц, а мы при поиске получили стоимость 1000, то поиск в этом направлении следует прекращать". Вот как выглядит псевдокод этого алгоритма (схема его работы представлена на рис. 12.27).



Порядок поиска: 1, 2, 3, ..., 17

Рис. 12.27. Поиск вглубь

```
void DepthFirstSearch(NODE r)
{
  NORE s;  // Используется при сканировании

  // Посещаем и помечаем корневой узел
  Visit(r);
  Mark(r);

  // Выполняем сканирование всех смежных узлов
  // по направлению от корня
  while( Имеется не посещенная вершина s, смежная с r)
  {
    DepthFirstSearch(s);
  } // while
} // DepthFirstSearch
```

Поиск Дейкстры

Этот вид поиска произошел от алгоритма, использующегося для поиска минимального перекрывающего граф дерева. При каждой итерации алгоритм Дейкстры определяет кратчайший путь к очередной точке и затем продолжает работу уже исходя из этой точки[2].

[2] Более подробно алгоритм Дейкстры описан в разделе 6.3 книги А. Ахо, Д. Хопкрофт, Д. Ульман. *Структуры данных и алгоритмы* — М. : Издательский дом "Вильямс", 2000. — *Прим. ред.*

A*-поиск

Этот вид поиска подобен алгоритму Дейкстры, с тем отличием, что он использует эвристику, которая не только просматривает стоимость от стартового до текущего узла, но и оценивает стоимость пути от текущего узла до целевого, несмотря на то что при этом на пути остаются непосещенные узлы.

Конечно, используются как модификации описанных алгоритмов, так и составные алгоритмы, однако общее представление о состоянии дел вы получили. Детальное описание этих алгоритмов можно найти в множестве различных книг, так что для тех, кто всерьез заинтересуется этим вопросом, найти подходящую книгу не будет большой проблемой.

СОВЕТ

Для поиска наилучшего пути могут также использоваться генетические алгоритмы. Их применение в реальном времени практически невозможно, но эти алгоритмы определенно более естественны, чем любой из описанных здесь.

Сценарии в работе искусственного интеллекта

Теперь, когда вас уже можно считать почти что экспертом в области искусственного интеллекта, я вновь хочу поговорить о сценариях — но уже на новом уровне.

Мы уже видели, что при работе системы ИИ может использоваться простейший язык сценариев, который в рассмотренном нами случае состоял из пар [ИНСТРУКЦИЯ ОПЕРАНД], и виртуальный интерпретатор такого языка. Однако такой простейший язык — это только начало, и предела совершенствованию языков сценариев нет. В качестве примеров высокоуровневых языков сценариев можно привести QUAKE C или UNREAL Script, которые позволяют разрабатывать код искусственного интеллекта игры с помощью высокоуровневого, похожего на естественный языка программирования, который затем обрабатывается игровым процессором.

Разработка языка сценариев

В основе разработки языка сценариев лежит та функциональность, которую вы хотите ему дать. При этом вы можете задать себе ряд вопросов.

- Будет ли язык сценариев использоваться только для искусственного интеллекта или его планируется использовать во всей игре?
- Будет ли данный язык компилироваться или интерпретироваться?
- Будет ли это язык сверхвысокого уровня, синтаксически напоминающий естественный язык, или это должен быть язык низкого уровня наподобие языков программирования с функциями, переменными, условными операторами и т.п.?
- Будет ли *весь* сценарий игры создаваться с использованием данного языка?
- Какой уровень сложности и мощности предполагается предоставить дизайнерам игры? Будут ли они иметь доступ к системным переменным и процессору игры?
- Каков профессиональный уровень дизайнеров, которые будут использовать разрабатываемый язык сценариев?

Именно над этими вопросами следует подумать, прежде чем приступать к созданию языка. И только после этого можно начинать непосредственную реализацию языка (да и всей игры).

Это очень важная фаза работы над игрой. Если планируется, что вашей игрой будут полностью управлять сценарии, просто необходимо, чтобы язык был надежным, мощным и расширяемым. Например, один и тот же язык сценариев должен быть пригоден как для моделирования летательных аппаратов, так и для моделирования поведения атакующих игрока монстров.

В любом случае запомните: основная идея языка сценариев — это создание высокоуровневого интерфейса к игровому процессору, наличие которого избавит вас от необходимости программировать управление объектами игры на низком уровне языка C/C++. Вместо этого для описания действий игры используется интерпретируемый или компилируемый псевдоестественный язык (рис. 12.28).

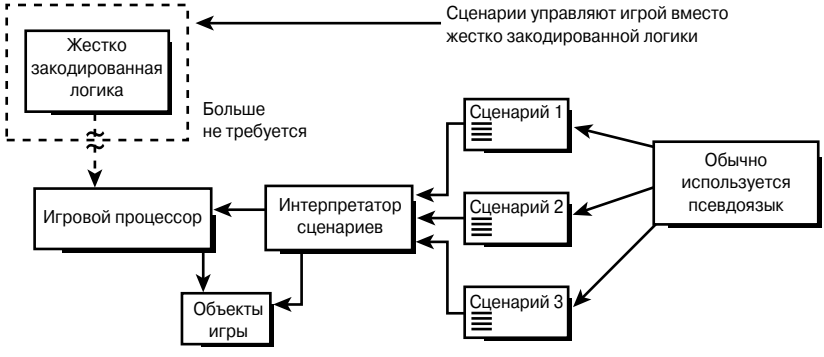


Рис. 12.28. Взаимосвязь игрового процессора и языка сценариев

Вот, например, сценарий на некотором воображаемом языке, управляющий уличным светофором:

```
OBJECT: "Street light", SL

VARIABLES:

  TIMER green_timer; // Для отслеживания времени

ONCREATE()            // Вызывается при создании объекта
BEGIN
  // Указываем цвет
  SL.ANIM = "GREEN"
END


// Основной сценарий
BEGINMAIN

  // Если кто-то находится вблизи светофора
  IF EVENT("near","Street light") AND
    TIMER_STATE(green_timer) EQUAL OFF THEN
    BEGIN
      SL.ANIM = "RED"
      START_TIMER(green_timer,10)
    ENDIF
```

```
  // Если время таймера истекло
  IF EVENT(green_timer,10) THEN
    BEGIN
      SL.ANIM = "GREEN"
    ENDIF

ENDMAIN
```

Не обращайте внимания ни на конкретный синтаксис примера, ни на действия сценария. Я хотел показать только его высокоуровневость, скрывающую множество деталей более низкого уровня — анимацию, проверку близости объекта к светофору и др. Однако не надо быть высококвалифицированным программистом, чтобы с помощью такого языка описать уличное движение.

Код начинается с создания светофора и указания его цвета (зеленый). Далее с помощью EVENT() идет проверка, не находится ли поблизости от светофора какой-либо объект, и если да, то светофор становится красного цвета, который через некоторое время автоматически сменяется зеленым. Обратите внимание: используемый язык выглядит похожим одновременно на C, BASIC и Pascal!

Разрабатываемый язык может быть не похож ни на один из известных языков программирования — дело ваше, но в таком случае ответственность за его работу и обучение работе с ним также возлагается на вас. Особенностью языка сценариев является то, что он должен знать, как работать с любым объектом игры. Например, когда вы используете вызов BLOWUP("whatever"), интерпретатор (или компилятор) языка должен знать, как взорвать любой из объектов, имеющихся в игре. Несмотря на то что взорвать синего монстра можно вызовом TermBMs3(), а взрыв стены осуществляется вызовом PolyFractWallOneSide(), вы можете просто написать BLOWUP("BLUE") или BLOWUP("wall").

Вероятно, вас интересует, как же создаются такие замечательные языки? Это вовсе не так просто. Для начала вам следует решить, будет ли язык интерпретируемым или компилируемым, т.е. будет ли сценарий на нем скомпилирован в непосредственный машинный код, или будет интерпретироваться игровым процессором в ходе игры, или будет использован некоторый промежуточный вариант. Затем вы должны разработать язык, лексический анализатор, генератор и интерпретатор промежуточного кода либо генератор машинного кода (а возможно, генератор кода на языке C/C++). Все эти вопросы вам помогут решить соответствующие инструменты, например такие, как LEX и YACC, предназначение которых — упростить реализацию лексического и синтаксического анализаторов для создания компилятора или интерпретатора языка сценариев. Это "взрослое" решение вопроса[3], а я сейчас предложу вам "детский" вариант, при котором вам не понадобится ничего, кроме препроцессора вашего компилятора C/C++.

Использование компилятора C/C++

Одним из основных достоинств интерпретируемых языков сценариев является возможность чтения и выполнения сценария без перекомпилирования всей игры. Если же этот вопрос не очень заботит вас, можете прибегнуть к одному старому трюку: использовать для трансляции вашего языка сценариев препроцессор C/C++.

Препроцессор C/C++ — очень мощный инструмент. Он позволяет решить множество задач на текстовом уровне, включая такие, как математические вычисления, подстановка

[3] В этом вам поможет соответствующая литература, например уже упоминавшаяся книга А. Ахо, Р. Сети, Д. Ульман. *Компиляторы: принципы, технологии и инструменты* — М. : Издательский дом "Вильямс", 2001. — *Прим. ред.*

строк, сравнения и многое другое. Разумеется, в конечном итоге вы получаете текст на языке C/C++, но ведь вы можете и не сообщать об этом дизайнеру игры, не так ли?

Наверное, наилучший способ разобраться, как использовать препроцессор для создания языка программирования сценариев — это привести конкретный очень простой пример (на большее у меня просто нет ни времени, ни сил).

Итак, язык сценариев будет компилируемым языком и каждый сценарий будет выполняться при создании описываемого им объекта. Выполнение сценария будет прекращаться, как только описываемый объект погибнет. Разрабатываемый язык сценариев основан на C, так что я не буду много говорить обо всех его тонкостях. Упомяну только, что для множества новых ключевых слов и типов данных используется подстановка текста.

Сценарий состоит из следующих частей.

Глобальный раздел. В нем определяются все глобальные переменные, использующиеся в сценарии. У нас есть только два типа данных: REAL и INTEGER. Тип REAL используется для хранения действительных чисел типа float, а INTEGER соответствует типу int.

Раздел функций. Этот раздел содержит определения функций, которые имеют следующий синтаксис:

```
data_type FUNCNAME(data_type parm1, data_type parm2 ...)
BEGIN
  // Код
ENDFUNC
```

Основной раздел. Это основная часть программы, которая циклически выполняется все время существования объекта в игре.

```
BEGINMAIN
  // Код
ENDMAIN
```

Что касается присвоения и сравнения, то возможно использование только следующих операторов:

Присвоение	variable = expression
Равенство	(expression EQUALS expression)
Неравенство	(expression NOTEQUAL expression)

Прочие типы сравнения используют тот же синтаксис, что и в языке C, т.е. имеют вид

```
(expression >  expression)
(expression <  expression)
(expression >= expression)
(expression <= expression)
```

Условные выражения имеют тот же вид, что и в C, с тем отличием, что код, выполняющийся при истинности условия, заключен в блок BEGIN ENDIF, а код, выполняющийся при ложности условия, — в блок BEGIN ENDELSE. Вот пример условного оператора:

```
if (a EQUALS b)
  BEGIN
    // Код
  ENDIF
else
  BEGIN
    // Код
  ENDELSE
```

Инструкция switch в таком языке отсутствует; есть только один тип цикла — WHILE, который имеет следующий синтаксис:

```
WHILE( condition )
  BEGIN
    // Код
  ENDWHILE
```

Имеется также ключевое слово GOTO для указания безусловного перехода из одной точки кода в другую. Метка точки назначения должна представлять собой ее имя с последующим двоеточием.

```
LBL_NAME:
```

Здесь часть имени NAME может представлять собой строку длиной до 16 символов, например:

```
LBL_DEAD:

if (a EQUALS b)
  BEGIN
    GOTO LBL_DEAD;
  ENDIF
```

Разумеется, вы можете добавить множество вспомогательных высокоуровневых функций для выполнения различных тестов состояния объектов. Например, для тех объектов, которым свойственно понятие здоровья, может быть разработана и использоваться функция HEALTH():

```
if (HEALTH("alien1") > 50)
  BEGIN
    // Код
  ENDIF
```

Кроме того, вы можете создавать события, которые могут использоваться для выполнения различных проверок в условных операторах и которые в качестве параметров принимают текстовые строки.

```
if (EVENT("player dead"))
  BEGIN
    // Код
  ENDIF
```

Это достигается путем использования глобальных переменных состояния и предварительной разработки достаточного количества событий для их применения в сценариях. Главным же оказывается подстановка текста с помощью препроцессора C/C++. Схема использования такого языка, где каждый файл сценария сначала обрабатывается препроцессором, показана на рис. 12.29.

Теперь с помощью препроцессора мы можем выполнить ряд подстановок в тексте сценариев и получить код на языке C/C++. Для работоспособности такой схемы все файлы сценариев сохраняются с расширением .SCR или каким-то другим предопределенным расширением, после чего перед компиляцией в файл сценария включается стандартный заголовочный файл трансляции сценариев. Вот как может выглядеть этот файл для только что рассмотренного языка:

```
// Типы переменных
#define REAL    static float
#define INTEGER  static int
```

```
// Сравнения
#define EQUALS  ==
#define NOTEQUAL !=

// Начала и концы блоков
#define BEGIN   {
#define END     }
#define BEGINMAIN {

#define ENDIF   }
#define ENDWHILE }
#define ENDELSE  }
#define ENDMAIN  }

// Цикл
#define WHILE   while
#define GOTO    goto
```

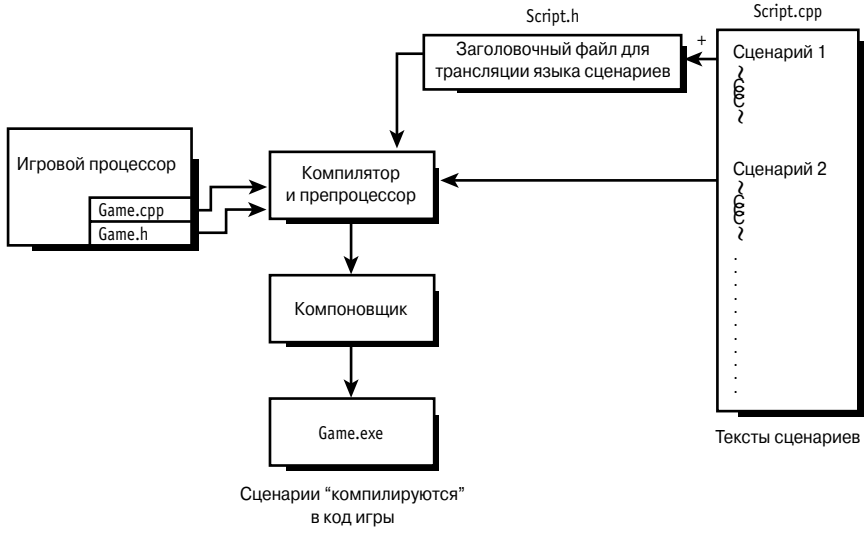


Рис. 12.29. Использование препроцессора C/C++ для интерпретации языка сценариев

Включение заголовочного файла трансляции сценариев выполняется с помощью директивы препроцессора

```
#include "SCRIPTTRANS.H"
```

Затем вы включаете файл сценариев в ваш код в некотором соответствующем разработанному сценарию месте; это может быть сделано как в начале кода, так и в другом месте — вплоть до включения в код некоторой функции.

```
Main_Game_Loop()
{
  #include "script1.scr"

  // Код функции
}  // Main_Game_Loop
```

Итак, код, написанный разработчиком сценариев, после трансляции препроцессором превращается в код на языке C/C++, который должен компилироваться вместе с прочим кодом игры и подобно прочему коду на C/C++ иметь возможность обращения к глобальным переменным, проверки осуществления тех или иных событий, вызова различных функций. Вот пример преобразования простейшего сценария в код C/C++:

```
// Переменные
INTEGER index;

index = 0;

// Основная часть кода
BEGINMAIN

LBL_START:

if (index EQUALS 10)
  BEGIN
    BLOWUP("self");
  ENDIF

if (index < 10)
  BEGIN
    index = index + 1;
    GOTO LBL_START;
  ENDIF

ENDMAIN
```

После обработки данного кода препроцессором мы получим обычный код на языке C/C++. Понятно, что для того, чтобы этот код был работоспособен, должна быть определена функция BLOWUP().

```
// Переменные
static int index;

index = 0;

// Основная часть кода
{

LBL_START:

if (index == 10)
  {
    BLOWUP("self");
  }

if (index < 10)
  {
    index = index + 1;
    goto LBL_START;
  }

}
```

Разумеется, при этом остается множество вопросов, которые следует решить и которые я не рассматриваю здесь даже бегло. Это и проблема коллизий имен, и доступа к глобальным переменным, и превращающаяся в настоящий кошмар при использовании препроцессора проблема отладки, и проблема проверки корректности кода сценария (например, отсутствия в нем бесконечных циклов), и многие другие. Тем не менее, думаю, основную идею использования существующего компилятора для работы со сценариями вы уловили.

СОВЕТ

Компилятор Visual C++ для вывода файла после обработки его препроцессором имеет опцию `/P`.

Искусственные нейронные сети

Нейронные сети — это одна из тех вещей, о которых все слышали, но никто не видел. Тем не менее в этой области в последние годы произошли существенные сдвиги. Здесь я расскажу о нейронных сетях очень кратко (если это вообще можно назвать рассказом) и не в контексте практического применения в играх, а просто для того, чтобы вы имели представление о том, что это такое.

Нейронная сеть представляет собой модель человеческого мозга. Мозг содержит от 10 до 100 миллиардов клеток, каждая из которых может обрабатывать и пересылать информацию. На рис. 12.30 показана модель клетки человеческого мозга (нейрона).

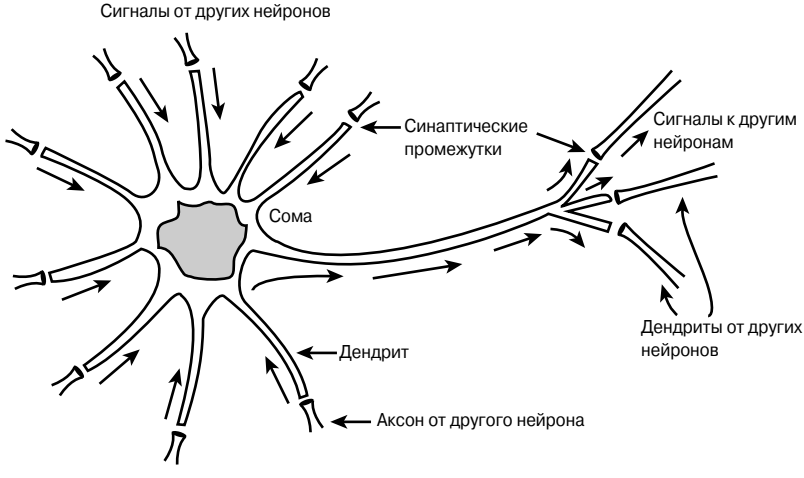


Рис. 12.30. Нейрон

Основными частями нейрона являются сома, аксон и дендриты. Сома представляет собой основное тело клетки и выполняет обработку сигналов, в то время как аксон передает сигналы дендритам, доставляющим их другим нейронам.

Каждый нейрон выполняет очень простую функцию: обработать входной сигнал и послать или не послать выходной сигнал. Нейроны имеют множество входов, единый выход (который может быть разделен) и некоторое правило, в соответствии с которым обрабатываются входные и генерируются выходные сигналы. Правила обработки чрезвычайно сложны; достаточно лишь сказать, что сигналы некоторым образом суммируются и полученный результат приводит к передаче нейроном выходного сигнала.

Искусственные нейронные сети представляют собой простые модели, которые, как и мозг, в состоянии обрабатывать информацию параллельно. Рассмотрим основные типы искусственных нейронов и нейродов.

Первая искусственная нейронная сеть была создана в 1943 году Мак-Каллохом (McCulloch) и Питтсом (Pitts), двумя инженерами, которые хотели смоделировать человеческий мозг с помощью электроники. Их модель основана на схеме, которую они назвали нейродом и которая показана на рис. 12.31 слева. К настоящему времени нейрод не очень изменился и выглядит теперь так, как показано на рис. 12.31 справа.

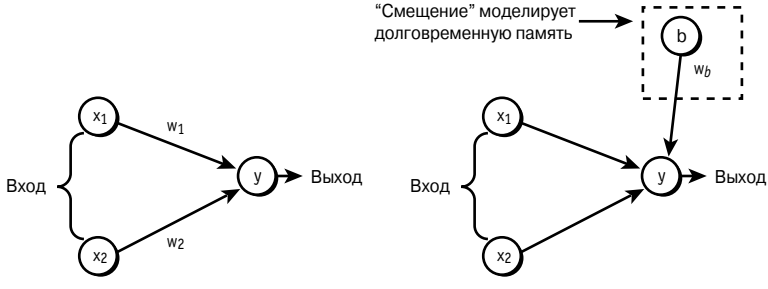


Рис. 12.31. Искусственный нейрон — нейрод

Нейрод состоит из ряда входов X_i, которые суммируются с учетом весов w_i, а затем обрабатываются с помощью функции активизации. Эта функция может быть как простой пороговой функцией (использовавшейся в модели Мак-Каллоха–Питтса), так и более сложной, например линейной или экспоненциальной. В пороговой модели суммарное значение сигналов сравнивается с пороговым значением и нейрон отправляет свой сигнал в том случае, когда суммарное значение превышает пороговое. Математически суммарное значение входных сигналов можно выразить следующим образом: $Y = \sum_{i=1}^{n} X_i w_i$. Соответственно выходной сигнал в пороговой модели будет посылаться только в том случае, когда значение Y превысит пороговое. В модели со смещением суммарное значение определяется формулой $Y = bw_b + \sum_{i=1}^{n} X_i w_i$.

Для того чтобы воочию увидеть работу нейрода, предположим, что у нас есть нейрод с двумя входными сигналами X_1 и X_2, которые могут принимать только одно из двух значений — 0 или 1. Установим порог равным 2, а веса сигналов $w_1 = w_2 = 1$. Выходной сигнал при разных входных сигналах для этой схемы нейрода показан в табл. 12.3. Как видите, данный нейрод просто реализует логический оператор `AND`. При помощи соответствующих функций активизации и весов сигналов нейроды могут легко реализовать вычисление любой логической функции. Например, на рис. 12.32 показаны модели для вычисления операторов `AND`, `OR` и `XOR`.

Разумеется, реальные нейронные сети гораздо сложнее, имеют множество слоев, сложные функции активизации, а количество нейродов в них может исчисляться тысячами. Но по крайней мере теперь вы представляете, как выглядят блоки, из которых строятся нейронные сети. Осталось только суметь применить их в играх — и мы получим искусственный интеллект, способный к самообучению и самостоятельному принятию решений.

Таблица 12.3. Таблица истинности простого нейрода

X_1	X_2	*Суммарный сигнал*	*Выходной сигнал*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

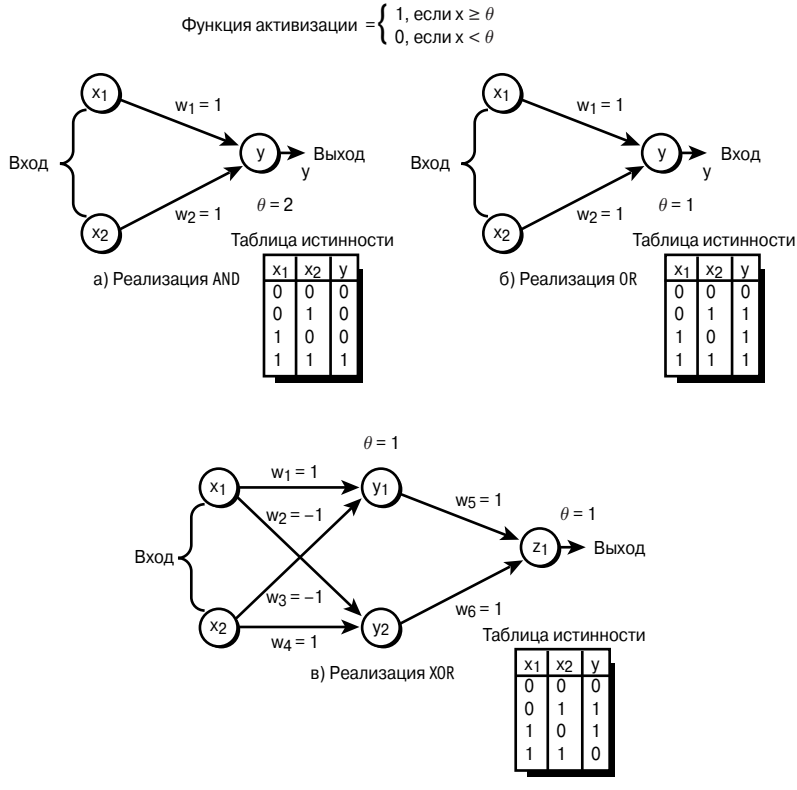


Рис. 12.32. Реализация логических операторов

Это очень интересная тема, но, увы, ограниченный размер книги не позволяет рассмотреть ее подробнее. Поэтому для тех, кто заинтересовался этим материалом, на прилагаемом компакт-диске в каталоге ARTICLES\NETWARE\ имеется более подробная статья по нейронным сетям, в которой рассматриваются различные типы сетей, описываются разные алгоритмы обучения, приводятся соответствующие исходные тексты программ и иллюстрации.

Генетические алгоритмы

Генетические алгоритмы представляют собой метод вычислений, основанный на биологических моделях эволюции. Они пытаются применить такие концепции, как естественный (и искусственный) отбор и генетические мутации в компьютерных моделях, призванные помочь решить задачи, которые не могут быть разрешены стандартными вычислительными средствами.

Принцип работы генетических алгоритмов примерно следующий. Мы представляем получаемую информацию в виде битового вектора, напоминающего ДНК (рис. 12.33).

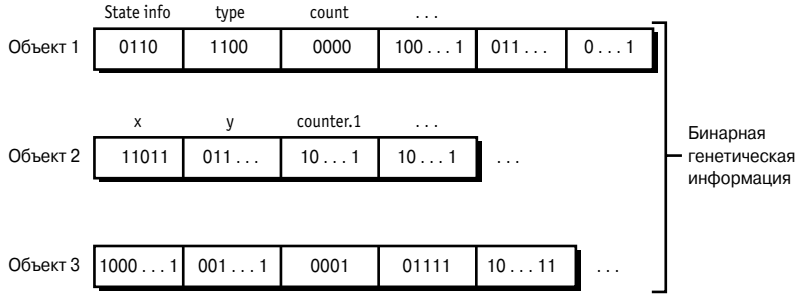


Рис. 12.33. Бинарное представление генетической информации

Этот битовый вектор представляет собой стратегию или код алгоритма решения задачи. Для начала нам требуется некоторое количество таких битовых векторов. Затем мы обрабатываем битовую строку и получаем с помощью некоторой целевой функции степень соответствия этой строки поставленной цели. Начальные битовые векторы могут быть получены на основании некоторых априорных данных или даже интуитивных представлений.

После обработки мы сравниваем степени соответствия исходных векторов поставленной задаче, а далее в действие вступает генетический алгоритм. Мы отбираем те векторы, которые дают наилучшее приближение к решению данной задачи, и моделируем их скрещивание, например так, как показано на рис. 12.34.

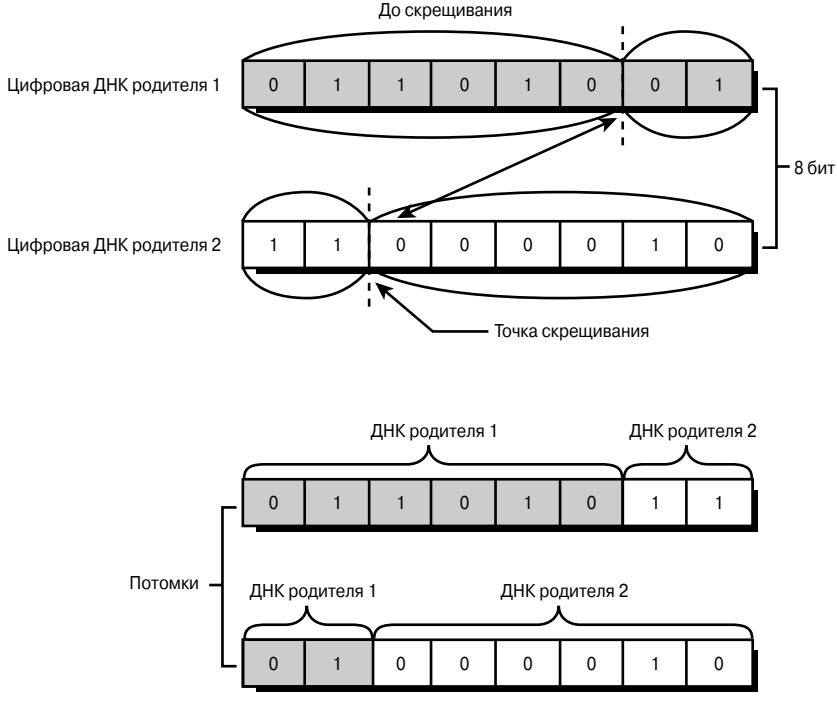


Рис. 12.34. Цифровое скрещивание

Для добавления небольшой непредсказуемости в процессе скрещивания некоторые биты могут быть изменены (моделирование мутаций). Получив новое поколение решений, мы испытываем их пригодность наряду с предыдущим поколением и отбираем для дальнейшего размножения только лучшие экземпляры. Это и есть процесс генетической эволюции. Наилучшее решение эволюционирует слабо, и, что интересно, при таком методе зачастую получаются результаты, которые порой даже невозможно себе представить.

Ключевым в идее генетических алгоритмов является то, что область поиска решения оказывается очень широкой. Это обычно никогда не достигается при поиске решения шаг за шагом, а достигается, помимо прочего, моделированием мутаций, представляющих собой совершенно случайные события, которые могут привести (или не привести) к лучшему соответствию битового вектора решению поставленной задачи.

Каким образом генетические алгоритмы могут быть использованы в играх? Миллионом способов. Один из них я вам открою, чтобы было с чего начать. В качестве цифровой ДНК можно использовать вероятностные установки искусственного интеллекта, а затем рассматривать в качестве целевой функции выживаемость персонажей с тем или иным вероятностным поведением, постепенно "выращивая" поколения персонажей, все более приспособленных к данной игре. Понятно, что вносить изменения в ДНК вы можете только при создании нового персонажа; есть и другие ограничения, но общая идея, надеюсь, вам понятна.

Нечеткая логика

Нечеткая логика (fuzzy logic) — последняя и, на мой взгляд, наиболее интересная технология, о которой я хочу вам рассказать. Она призвана делать выводы на основе теории нечетких множеств. Другими словами, нечеткая логика — метод анализа множеств данных, таких, что их элементы могут входить во множества частично. Мы в основном используем четкую логику, где элемент либо входит в множество, либо нет. Например, если у меня есть множества детей и взрослых, то я попадаю во множество взрослых, а мой трехлетний племянник — во множество детей. Это и есть четкая логика.

Нечеткая же логика позволяет объектам содержаться в множествах, даже если они не находятся в них полностью. Например, я могу сказать, что на 10% принадлежу множеству детей и на 100% — множеству взрослых. Это и есть нечеткие значения. Аналогично, о племяннике можно сказать, что на 2% он принадлежит множеству взрослых, и на 100% — множеству детей. Заметим также, что нечеткие значения не составляют в сумме 100%; эта сумма может быть как больше, так и меньше 100%. Дело в том, что это не вероятности, а степени включения в различные классы. Тем не менее сумма вероятностей принадлежности различным классам должна быть равна 1.0.

Самое ценное в нечеткой логике то, что она позволяет на основе нечетких или не совсем верных данных принимать (как правило) верные решения. Четкая логика этого не позволяет: если у вас отсутствует некоторая переменная или входные данные, дальнейшая работа оказывается просто невозможной. Однако нечеткая логика в состоянии продолжать работу и в этом случае, как и человеческий мозг. (Я, кстати, иногда задумываюсь: как много решений мы принимаем на основании нечеткой логики? Как правило, у нас никогда нет полных данных, и в то же время мы почти всегда уверены в корректности своих решений.)

Возможности применения нечеткой логики в искусственном интеллекте в областях принятия решений, фильтрации ввода-вывода, выборе поведения должны быть очевидны. Рассмотрим теперь, каким образом можно реализовать и использовать нечеткую логику.

Теория обычных множеств

Обычное множество представляет собой просто набор объектов. Для записи множества используют строчную букву, которая представляет его имя, а затем помещают содержащиеся в нем элементы в фигурных скобках, разделяя их запятыми. Множества могут состоять из чего угодно: имен, чисел, цветов... На рис. 12.35 показано несколько обычных множеств.

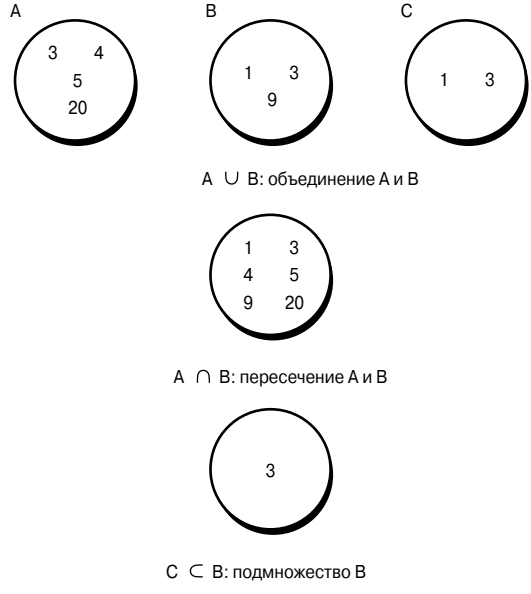


Рис. 12.35. Некоторые простые множества

Например, множество $A=\{3,4,5,20\}$ и множество $B=\{1,3,9\}$. Существует целый ряд операций, которые могут быть выполнены над множествами.

Является элементом ($\in$). Когда речь идет о некотором множестве, может понадобиться узнать, содержится ли некоторый элемент в данном множестве, т.е. связан ли данный элемент с множеством отношения включения. В нашем примере 3 является элементом *A*, а 2 не является элементом *B*, что записывается как $3\in A$ и $2\notin B$.

Объединение ($\cup$). Этот оператор собирает все объекты, имеющиеся в двух множествах, и создает из них новое множество. Если некоторый элемент содержится в обоих множествах, он добавляется в новое множество только один раз. В нашем примере $A\cup B=\{1,3,4,5,9,20\}$.

Пересечение ($\cap$). Этот оператор создает новое множество только из тех элементов, которые имеются в обоих множествах одновременно. Таким образом, $A\cap B=\{3\}$.

Является подмножеством ($\subset$). Когда речь идет о некотором множестве, может понадобиться узнать, содержится ли в нем некоторое другое множество целиком (т.е. все элементы одного множества одновременно являются элементами другого множества). Если да, то множество, все элементы которого содержатся в другом множестве, является подмножеством этого множества. Таким образом, множество $\{1,3\}$ является подмножеством *B*, что записывается как $\{1,3\}\subset B$, но множество *A* не является подмножеством *B*: $A\not\subset B$.

Вот немного азов теории множеств. Ничего сложного — только чуть-чуть терминологии и обозначений. Каждый из нас сталкивается с этой теорией ежедневно, мы просто не знаем об этом. Главное, что вам следует запомнить, — это то, что обычные множества являются точными. Нечто может либо быть фруктом, либо нет. Пятерка либо имеется в некотором множестве, либо нет. Однако это не так в случае нечетких множеств.

Теория нечетких множеств

Проблема с компьютерами состоит в том, что эти точные по своей природе машины пытаются использовать для решения неточных или нечетких задач. В 1970-х годах в программировании и решении вычислительных задач ученые начали применять математические методы, именуемые *нечеткой логикой*. Рассмотрим все то же, что только что было рассказано о теории обычных множеств, но теперь для нечетких множеств.

В теории нечетких множеств мы не смотрим на то, сколько именно объектов принадлежит множеству. Эти объекты находятся в множестве, но нас интересует степень принадлежности конкретного объекта определенному классу. Создадим, например, нечеткий класс *Компьютерных спецэффектов*, а затем возьмем несколько фильмов и оценим, насколько каждый из них соответствует этому классу (табл. 12.4).

Таблица 12.4. Степень принадлежности классу "*Компьютерные спецэффекты*"

Фильм	*Степень принадлежности классу, %*
Antz	100
Forrest Gump	20
The Terminator	75
Aliens	50
The Matrix	90

Вы видите, насколько все здесь неточно? Хотя *The Matrix* имеет множество компьютерных спецэффектов, а фильм *Antz* целиком создан с помощью компьютеров, так что в чем-то я прав. Но согласны ли вы с конкретными числами? Продолжительность сгенерированного компьютером фильма *Antz* — 1 час 20 минут, а в фильме *Forrest Gump* все спецэффекты в сумме занимают 5 минут. Так насколько честно говорить о том, что степень принадлежности классу для этого фильма — 20%? Я не знаю, и именно поэтому использую нечеткую логику.

Как бы то ни было, но каждая нечеткая степень участия во множестве записывается в виде пары значений "{кандидат на включение, степень принадлежности}". Таким образом, в нашем примере мы должны записать "{Antz, 1.00}, {Forrest Gump, 0.20}, {The Terminator, 0.75}, {Aliens, 0.50}, {The Matrix, 0.90}". И наконец, как вы включите сегодняшний день в класс *Дождливо*? Там, где я живу, сегодняшний день стоит включить как "{Сегодня, 1.00}"!

Теперь добавим немного абстракции и создадим *полностью нечеткое множество*. В большинстве случаев это упорядоченное множество *степеней членства* (или *принадлежности*) (degree of membership — DOM) множества объектов определенного класса. Например, в классе *Компьютерные спецэффекты* мы имеем множество, составленное из степеней принадлежности $A = \{1.00, 0.20, 0.75, 0.50, 0.90\}$. Множество содержит по одной записи для каждого фильма: каждая переменная представляет DOM каждого фильма из табл. 12.4 (так что, как вы понимаете, порядок элементов множества очень важен).

Теперь предположим, что у нас есть другое множество фильмов, которые имеют свои степени принадлежности: $B = \{0.2, 0.45, 0.5, 0.9, 0.15\}$. Применим к описанным множествам некоторые из рассмотренных ранее операций и посмотрим, что у нас получится. Пе-

ред тем как приступить, позвольте заметить, что, поскольку речь идет о нечетких множествах, которые представлены степенями принадлежности или вектором соответствия множества объектов, при многих операциях с множествами последние должны иметь одинаковое количество элементов. Это требование станет более понятным, после того как мы рассмотрим операторы для работы с нечеткими множествами.

Нечеткое объединение ($\cup$). Объединение двух нечетких множеств представляет собой множество максимальных значений из каждого множества: $A \cup B = \{\max(a_i, b_i)\}$, например:

```
A = {1.00, 0.20, 0.75, 0.50, 0.90}
B = {0.20, 0.45, 0.50, 0.90, 0.15}
A ∪ B =
  {1.00, 0.45, 0.75, 0.90, 0.90}
```

Нечеткое пересечение ($\cap$). Пересечение двух нечетких множеств представляет собой множество минимальных значений из каждого множества: $A \cup B = \{\min(a_i, b_i)\}$, например:

```
A = {1.00, 0.20, 0.75, 0.50, 0.90}
B = {0.20, 0.45, 0.50, 0.90, 0.15}
A ∩ B =
  {0.20, 0.20, 0.50, 0.50, 0.15}
```

Подмножества и элементы в нечетких множествах имеют меньшее значение, чем в стандартных множествах, так что не будем рассматривать здесь соответствующие отношения. Однако для нечетких множеств есть еще одно отношение — дополнительности (комплементарности). Дополнением нечеткой переменной со степенью принадлежности x является (1-x), так что дополнением множества A является множество $A' = \{1 - a_i\}$, например:

```
A = {1.00, 0.20, 0.75, 0.50, 0.90}
A′ = {0.00, 0.80, 0.25, 0.50, 0.10}
```

Переменные и правила нечеткой лингвистики

Теперь, когда у вас есть представление об идее нечетких переменных и множеств, рассмотрим, как можно попытаться использовать их в системе искусственного интеллекта игры. Вы создадите механизм искусственного интеллекта, использующий нечеткие правила, применяющий нечеткую логику на входе и производящий нечеткий или четкий выход для контролируемых объектов игры.

Объединяя нормальные условные логические утверждения, вы создаете некоторое количество утверждений или дерево с предложениями вида

```
if X AND Y then Z
```

или

```
if X OR Y then Z
```

Переменные X и Y называются *предпосылками*, а Z — *следствием*. Однако в нечеткой логике X и Y — это нечеткие лингвистические переменные (fuzzy linguistic variables). Основным во всей этой нечеткой кухне является то, что X и Y нечеткие переменные. Нечеткие утверждения такой формы называются *правилами*, и в конечном счете вы можете вычислить их за несколько шагов. Их нельзя вычислять, как обычные логические утверждения типа

```
if EXPLOSION AND DAMAGE then RUN
```

Для нечеткой логики правила — это только части окончательного решения.

Нечеткие лингвистические переменные представляют нечеткую концепцию, которая оперирует с диапазонами. Например, предположим, что вы хотите классифицировать расстояние между игроками и объектами игры с искусственным интеллектом с помощью трех различных нечетких лингвистических переменных (по существу, названий). Посмотрите на рис. 12.36. На нем изображено нечеткое многообразие или поверхность, состоящая из трех различных треугольных областей, которые я назвал так:

NEAR	Область в диапазоне (от 0 до 306)
CLOSE	Область в диапазоне (от 250 до 700)
FAR	Область в диапазоне (от 500 до 1000)

Входные переменные изображены на оси *X* и могут изменяться от 0 до 1000. Этот диапазон называется *областью определения*. Выход нечеткого многообразия находится на оси *Y* и изменяется от 0.0 до 1.0. Для различных входных величин x_i (в этом примере они представляют расстояние до игрока) вы определяете степень принадлежности, проводя вертикальную линию, как показано на рис. 12.37, и подсчитывая значение (значения) *y* на пересечении с каждой лингвистической переменной треугольной областью.

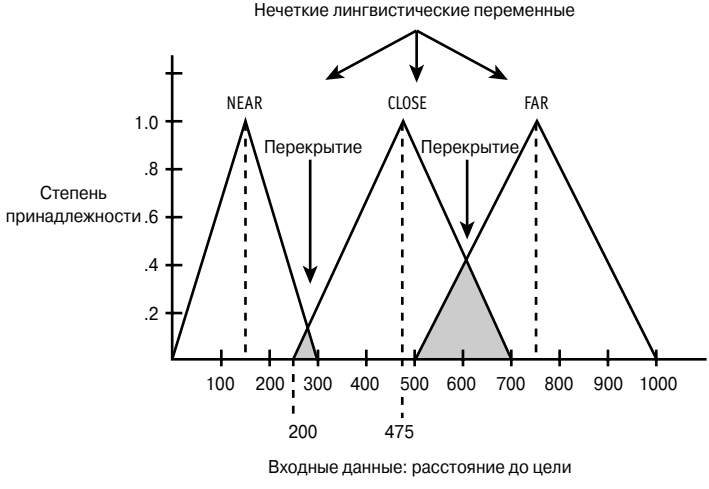


Рис. 12.36. Нечеткое многообразие, составленное из диапазонов лингвистических переменных

Каждый треугольник нечеткой поверхности представляет собой область влияния каждой нечеткой лингвистической переменной (NEAR, CLOSE, FAR). Области влияния перекрываются — обычно до 10–50%. Это связано с тем, что нет четких границ, когда ближайшее становится близким, а близкое — далеким. Нужны небольшие области пересечений, чтобы смоделировать нечеткость ситуации.

Сделаем некоторые выводы. У нас есть правила, которые основываются на нечеткой входной информации от игры, окружающей среды и т.д. Эти правила могут выглядеть так же, как и утверждения обычной условной логики, но они должны вычисляться с использованием нечеткой логики, поскольку на самом деле представляют собой нечеткие лингвистические переменные, классифицирующие входные данные по степени их принадлежности.

Окончательный результат работы нечеткой логики может как быть приведен к дискретным четким величинам, например к конкретным действиям объекта либо к кон-

кретным величинам тех или иных переменных, так и оставаться нечетким для очередных стадий работы нечеткой логики.

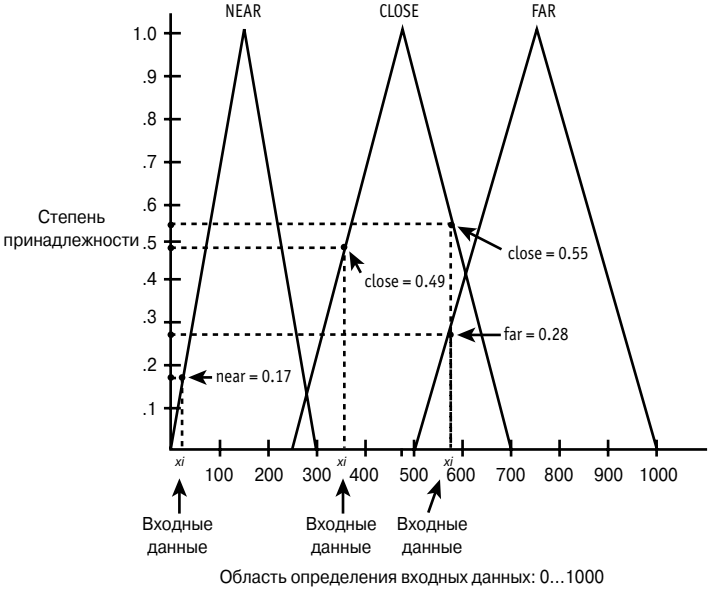


Рис. 12.37. Оценка степени принадлежности значения одной или нескольким переменным

Итак, теперь вы знаете, что у вас есть несколько входов в вашей нечеткой системе ИИ. Эти входы классифицируются одной или несколькими нечеткими лингвистическими переменными, представляющими некоторые диапазоны. Затем для каждого входа подсчитывается степень принадлежности каждому диапазону, как было показано ранее (см. рис. 12.37). Следует заметить, что наряду с треугольниками могут использоваться и другие формы, например трапеции или гауссовы кривые.

В любом случае степень принадлежности определяется по пересечению вертикальной линии для данного входа x_j и графика для каждой из нечетких лингвистических переменных. Линия может пересекать несколько графиков, и все эти пересечения должны быть вычислены при определении степеней принадлежности.

Нечеткая ассоциативная матрица

Нечеткие ассоциативные матрицы (fuzzy associative matrices) используются для получения результатов на основе нескольких нечетких входных данных и заданной базы правил. На рис. 12.38 такая матрица показана в графическом виде.

В большинстве случаев такие матрицы оперируют только с двумя нечеткими переменными. Каждая запись матрицы представляет собой логическое выражение типа if X_i AND Y_i then Z_i, где X_i — нечеткая логическая переменная на оси X, Y_i — нечеткая лингвистическая переменная на оси Y, а Z_i — выход, который может быть как нечеткой, так и четкой переменной.

Чтобы использовать нечеткие ассоциативные матрицы, поступим следующим образом. Возьмем четкий вход и для каждой нечеткой переменной подсчитаем его степень принадлежности. После этого обратимся к нечеткой ассоциативной матрице и проверим

правила в каждой ее ячейке, чтобы вычислить выходные значения на основе нечетких входных значений. При вычислении результатов применения правил применяются законы нечеткой логики, например использование функции MIN() при вычислении логического AND. После вычисления всех элементов матрицы нам нужно получить четкое значение, которое можно вычислить, либо используя максимальные из полученных значений, либо усредняя их.

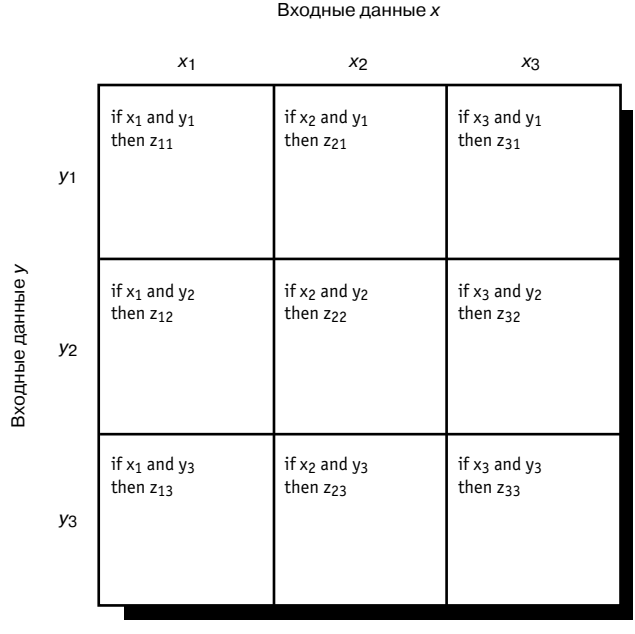


Рис. 12.38. Использование нечетких ассоциативных матриц

Конечно же, все рассказанное мною здесь — это не более чем отдельные намеки на то, что представляет собой нечеткая логика на самом деле. Но если вас заинтересовала эта тема, то в Internet вы сможете найти и теоретические материалы, посвященные ей, и множество различных программ — как коммерческих, так и нет.

Создание реальных систем ИИ для игр

Вот и все, что я хотел рассказать вам об азах использования искусственного интеллекта в играх. Я рассказал вам о некоторых используемых для этого технологиях исключительно для того, чтобы вам было с чего начать. Вы можете растеряться и не знать, какие технологии использовать в том или ином конкретном случае. Вот несколько советов на эту тему.

- Для объектов с простым поведением, таких, как камни, ракеты и т.п., используйте простейшие детерминированные системы ИИ.
- Для объектов, которые предполагаются разумными, но являются скорее частью окружающей среды, чем действующими объектами (например, летающие вокруг птицы, пролетающий рядом космический корабль), используйте детерминированный ИИ с элементами рандомизации.

- Для важных персонажей игр, взаимодействующих с игроками, вам, безусловно, необходимо использовать конечные автоматы вместе с другими вспомогательными методами. Однако некоторые персонажи могут быть не очень умными — в таком случае не следует затрачивать излишние усилия на их персонализацию посредством специализированных распределений вероятностей или терять время на разработку самообучающихся систем.
- При реализации очень умного персонажа игры вы должны собрать все вместе. Его ИИ должен быть конечным автоматом с применением большого количества условной логики, вероятностей, с запоминанием переходов между состояниями.

Не забывайте, что очень сложная система может состоять из массы очень простых составляющих. Другими словами, даже когда искусственный интеллект каждого индивидуального персонажа совсем несложен, их взаимодействие создает огромную поведенческую систему, запрограммировать которую "с нуля" представляется совершенно непосильной задачей (в чем-то это похоже на мозг человека — чрезвычайно сложную систему, состоящую из предельно простых нейронов). Очень важную роль в таких сложных системах играет обмен информацией между ее составляющими; в играх это реализуется как обмен информацией между оказавшимися на близком расстоянии персонажами.

Резюме

Эта глава посвящена очень интересной теме, не правда ли? Надеюсь, ваш интеллект оказался способен справиться с материалом об искусственном интеллекте и вы теперь получили представление о том, что такое конечные автоматы, деревья принятия решений, языки сценариев, нейронные сети, генетические алгоритмы и нечеткая логика. Конечно, я вынужден в который раз повториться — рамки этой книги слишком малы для такой всеобъемлющей темы, так что моей главной задачей было рассказать о том, что есть в этой области, чтобы вы знали, поисками чего следует заняться при написании той или иной игры. И все же я думаю, что никакой искусственный интеллект не может превзойти человеческий (иначе я бы уже давно предоставил ему возможность зарабатывать деньги вместо меня :-)).

ГЛАВА 13

Основы физического моделирования

Совершенно невозможно представить себе игры с использованием реальной физики в 1970–80-х годах. В большинстве своем это были игры типа “стрелялок”, где надо было найти и уничтожить всех врагов. Однако с приходом 1990-х годов и эры трехмерных игр физическое моделирование становится все более и более важным. Сейчас трудно представить себе игру, в которой движение объектов не реалистично или хотя бы не сделана попытка приблизить его к таковому. В этой главе представлены основные концепции физического моделирования.

- Фундаментальные законы физики
- Гравитация
- Трение
- Столкновения
- Кинематика
- Системы частиц

Если Вселенная — всего лишь модель на некотором невообразимо сложном компьютере, то Бог — ее программист. А значит, законы физики отлично работают на всех уровнях — от квантового до космогонического. Красота физики в том, что Вселенной управляют не так уж много законов физики. Другое дело — ограниченность наших знаний об этих законах; однако для создания компьютерной модели, способной обмануть почти любого наблюдателя своим реализмом, знаний у нас достаточно.

Большинство моделей и игр используют модели на базе стандартной классической физики Ньютона, которая вполне справедлива для описания движения объектов в разумных пределах масс, размеров и скоростей (т.е. скорость должна быть гораздо меньше скорости света, объекты — гораздо больше отдельного атома, но гораздо меньше галактики). Но даже такое моделирование на основе физики Ньютона требует немалых затрат компьютерных сил. Простая имитация дождя или бильярда, выполненная полностью корректно, потребует, как минимум, мощи Pentium IV.

Тем не менее мы не раз видели в играх и дождь и бильярд. Так в чем же дело? Программисты этих игр отдавали себе отчет в том, что физика, которую они должны смоделировать, слишком сложна и следует ограничиться очень приближенными моделями, дающими результаты, близкие к тем, с которыми игрок сталкивается в реальной жизни. В таких программах множество различных уловок, оптимизаций и упрощений моделируемой системы. Например, гораздо проще вычислить результат столкновения двух сфер, чем результат столкновения астероидов неправильной формы, поэтому программист может аппроксимировать астероиды в игре простыми сферами.

Не хватит и тысяч страниц, чтобы описать действительно серьезную адекватную физическую модель, так как в ней должна будет излагаться не только физика, но и математика, которую следует изучить, чтобы хотя бы понимать, о чем идет речь. Поэтому я ограничусь только самыми простыми физическими моделями, изучив которые, а затем освоив в необходимом объеме соответствующие разделы физики и математики, вы сможете создавать собственные модели для своих игр.

Фундаментальные законы физики

Начнем путешествие в мир физики с описания некоторых базовых концепций и свойств пространства, времени и материи. Эти концепции познакомят вас с используемой терминологией и помогут понять следующий за этим разделом более сложный материал.

ВНИМАНИЕ

Все, о чем я буду говорить, не вполне верно на квантовом или космологическом уровнях, но достаточно корректно на уровне нашего обсуждения. Разумеется, это не учебник физики и говорить о полной строгости в изложении материала здесь не приходится. В книге используется метрическая система (что непривычно только для читателей из США).

Масса (m)

Вся материя имеет массу, которая является характеристикой ее инертности и, как правило, определяет, какое количество атомов составляет данный объект. Заметим — я ничего не говорю о весе! Многие путают понятия массы и веса и говорят, что они весят на Земле 75 kg[1]. Начнем с того, что килограмм — это единица измерения массы, т.е. того, сколько материи содержит объект. Единицей измерения веса является *Ньютон* (*N*). Материя сама по себе не имеет веса; только будучи помещена в гравитационное поле, которое и вызывает эффект, называемый весом. Следовательно, масса представляет собой более абстрактную идею, чем вес, который изменяется от планеты к планете.

В играх концепция массы используется в большинстве случаев абстрактно, как относительная величина. Например, я могу указать, что масса корабля — 100 единиц, а масса астероида — 10000. Я могу использовать для описания массы килограммы, но пока я не приступаю к реальному физическому моделированию, никакого значения это не имеет.

[1] В книге используются международные обозначения единиц физических величин, в соответствии с требованиями SI. — *Прим. ред.*

Мне необходимо знать лишь то, что объект массой 100 единиц в два раза массивнее объекта с массой 50 единиц. Я еще вернусь к вопросу о массе позже, когда будут рассматриваться сила и гравитация.

НА ЗАМЕТКУ

Масса также представляет собой меру сопротивления объекта изменению его скорости (первый закон Ньютона). Первый закон Ньютона гласит, что в отсутствие внешних сил неподвижный объект остается в состоянии покоя, а движущийся — движется прямолинейно и с постоянной скоростью.

Время (t)

Время представляет собой абстрактную концепцию. Подумайте о том, как можно объяснить, что такое время, не прибегая к нему в процессе объяснения? Без цикличности в определении разъяснить, что такое время, не удастся. Но, к счастью, все и так знают, что это такое, а потому и говорить не о чем, разве что о том, как концепция времени связана с играми.

Время в реальной жизни измеряется в секундах, минутах, часах и т.д. Если вам нужно измерять время более точно — к вашим услугам миллисекунды (1 ms = 10^{-3}s), микросекунды (1 μs = 10^{-6}s), нано-, пико-, фемто- (10^{-9}, 10^{-12} и 10^{-15} соответственно) и прочие доли секунды. Однако в большинстве видеоигр никакой корреляции между реальным временем и временем в игре не наблюдается. Разрабатываемые алгоритмы в большей степени привязаны к частоте кадров, чем к реальному времени и секундам (за исключением игр, где моделируется реальное время). В большинстве игр один кадр — это и есть одна виртуальная секунда, другими словами, наименьшее различимое количество времени.

Однако при создании реальных сложных трехмерных игр вам, видимо, все же придется работать с реальным временем. Все алгоритмы в таких играх не должны привязываться к частоте кадров, например танк, идущий со скоростью 70 km/h, должен идти именно с этой скоростью, упадет ли частота кадров до двух в секунду или подскочит до 60... Моделирование времени в таких условиях — сущее наказание, но, если вы хотите создать реалистичную игру, без него невозможно обойтись и физические события не должны зависеть от изменения частоты кадров. В любом случае в наших примерах время будет измеряться в секундах (s) или в виртуальных секундах (которые просто означают смену одного кадра).

Положение в пространстве (s)

Каждый объект имеет определенное положение (x, y, z) в трехмерном пространстве (x, y) — в двухмерном, просто x (иногда используется обозначение s) — в одномерном (рис. 13.1). Однако иногда непонятно, что считать положением объекта, даже когда известно, где именно он находится. Например, выбирая точку, которая определяет положение сферы (рис. 13.2), вы наверняка выберете ее центр. Но что делать, если этот объект, например, молоток? Молоток представляет собой объект сложной формы, так что большинство физиков будут использовать его центр масс в качестве характеристики его положения в пространстве (рис. 13.3).

Однако в играх концепция положения в пространстве гораздо проще. Большинство программистов просто описывают вокруг объекта прямоугольник, окружность (параллелепипед, сферу, в зависимости от размерности игры), как показано на рис. 13.4, и используют центр описанного объекта в качестве центра рассматриваемого объекта. В большинстве случаев такое описание положения объекта вполне приемлемо, особенно когда основная масса объекта расположена в его центре; но при выполнении вычислений в соответствии с физической моделью это приближение является слишком грубым и неадекватным.

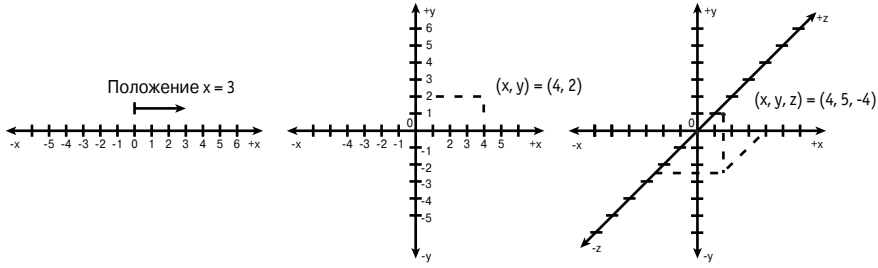


Рис. 13.1. Положение точки в пространстве

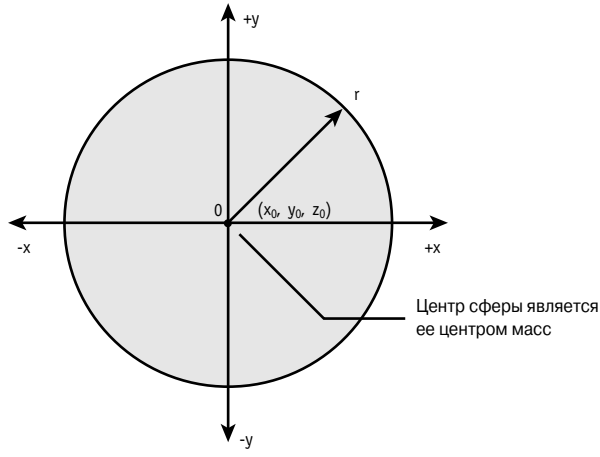


Рис. 13.2. Центр масс сферы

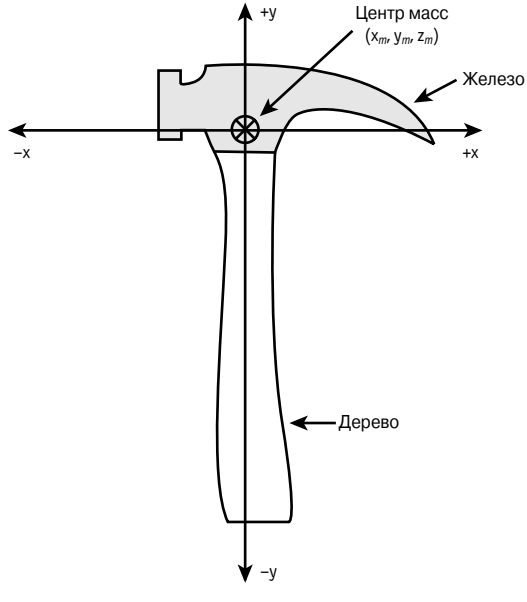


Рис. 13.3. Центр масс произвольного объекта

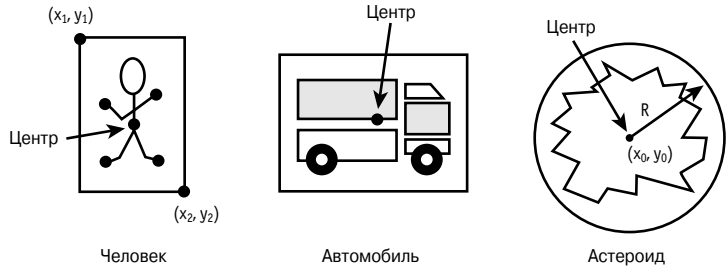


Рис. 13.4. Контуры столкновений

Единственный путь решения данной проблемы состоит в том, чтобы выбрать в качестве точки, представляющей положение объекта, его центр масс. Вычислить центр масс достаточно просто — нужно только разбить объект на элементарные малые по размеру ячейки и вычислить их вес (например, как количество точек объекта, попадающих в данную часть), а затем вычислить координаты центра масс по общей формуле

$$\vec{r}_c = \frac{\sum_i \vec{r}_i m_i}{\sum_i m_i},$$

где m_i — массы элементарных ячеек, а r_i — радиус-векторы центров масс этих ячеек.

Если же объект составлен из многоугольников, то каждому из них можно сопоставить его точный вычисленный вес и центр масс, и вычислить центр масс объекта в точности. Предполагая, что двухмерный объект состоит из n частей, масса i-й части при этом m_i, а координаты ее центра масс — (x_i, y_i), вычислить координаты центра масс объекта можно по формулам

$$x_c = \frac{\sum_{i=0}^{n-1} x_i m_i}{\sum_{i=0}^{n-1} m_i},$$

$$y_c = \frac{\sum_{i=0}^{n-1} y_i m_i}{\sum_{i=0}^{n-1} m_i}.$$

Обозначение $\sum_i f_i$ означает “сумма” и работает наподобие простого цикла, в котором выполняется суммирование значений f_i для каждого i.

Скорость (v)

Скорость представляет собой мгновенную меру поступательного движения объекта и измеряется как отношение пройденного объектом расстояния ко времени, за которое оно было пройдено. Единица измерения скорости — m/s. Математически скорость представляет собой первую производную по времени радиус-вектора движущейся точки:

$$\vec{v} = \frac{\mathrm{d}\vec{r}}{\mathrm{d}t},$$

а в одномерном случае:

$$v = \frac{ds}{dt}.$$

Другими словами, скорость — это отношение мгновенного изменения положения точки в пространстве к затраченному на это времени.

В видеоиграх концепция скорости одна из наиболее распространенных, однако используемые при этом единицы, как правило, достаточно произвольны и относительны. Так, в ряде демонстрационных программ я обычно перемещал объекты на несколько пикселей за кадр при помощи следующего кода:

```
x_position += x_velocity;
y_position += y_velocity;
```

Однако вы помните, что кадры временем не являются? Их можно считать виртуальным временем только до тех пор, пока скорость вывода кадров остается неизменной. Полагая, что она равна 30 кадров в секунду, а перемещение — 4 пикселя за кадр, можно рассчитать виртуальную скорость перемещения объекта, равную

```
4 пикселя/(1/30 секунды) = 120 пикселей в секунду
```

Следовательно, объект в нашей игре перемещается со скоростью, измеряемой в пикселях в секунду. Конечно, если хотите, то можете оценить, сколько метров содержится в одном пикселе вашего виртуального игрового пространства, и выразить полученную скорость в привычных метрах в секунду. В любом случае, зная скорость объекта и его положение в одном кадре, вы можете вычислить, какое положение он будет занимать в следующем кадре. Так, если в настоящий момент положение объекта x_0 и он движется с постоянной скоростью 4 пикселя за кадр, то через 30 кадров его положение будет $x_0 + 4 \times 30 = x_0 + 120$ пикселей. Итак, главная формула поступательного движения с постоянной скоростью: $x_t = x_0 + vt$. Эта формула гласит, что если объект, расположенный в точке x_0, движется с постоянной скоростью v в течение времени t, то к этому моменту его положение будет равно положению в начальный момент плюс произведение скорости на время движения vt (рис. 13.5). В демонстрационной программе `DEMO13_1.CPP` на прилагаемом компакт-диске каждый вертолет движется по экрану слева направо с постоянной скоростью.

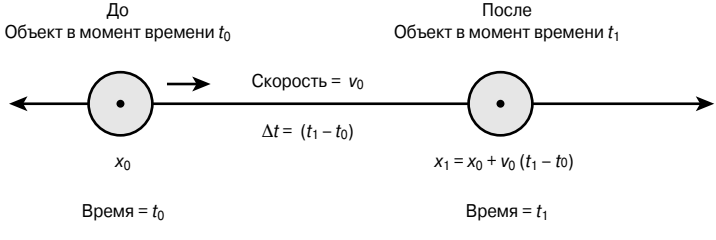
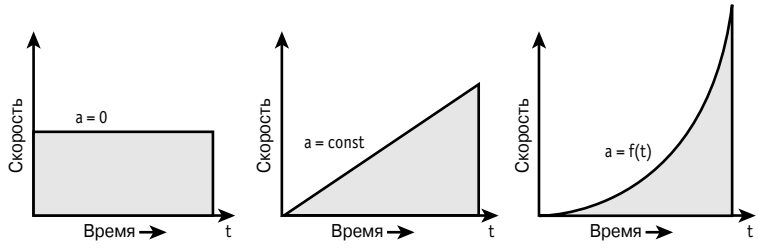


Рис. 13.5. Движение с постоянной скоростью

Ускорение (a)

Ускорение похоже на скорость, но определяет скорость изменения скорости. Взгляните на рис. 13.6 — на нем проиллюстрированы объект, движущийся с постоянной скоростью, и объекты, движущиеся ускоренно. Графиком зависимости скорости от времени для объекта, движущегося с постоянной скоростью, будет горизонтальная прямая. Для объекта, движущегося с ускорением, зависимость скорости от времени представляет собой наклонную прямую (при постоянном ускорении) или кривую.



а) Постоянная скорость б) Постоянное ускорение в) Переменное ускорение

Рис. 13.6. Скорость и ускорение

Математически ускорение определяется так же, как и скорость, с той разницей, что радиус-вектор объекта заменяется вектором его скорости:

$$\vec{a} = \frac{\mathrm{d}\vec{v}}{\mathrm{d}t};$$

в одномерном случае соответственно

$$a = \frac{\mathrm{d}v}{\mathrm{d}t}.$$

Единица измерения ускорения, как видно из его определения, — m/s^2. Одномерное равноускоренное прямолинейное движение дает следующие формулы для зависимости скорости и положения объекта от времени (в момент времени $t = 0$ объект находился в положении x_0, а его скорость составляла v_0; ускорение объекта постоянно и равно a):

$$v_t = v_0 + at,$$
$$x_t = x_0 + v_0 t + \frac{at^2}{2}.$$

Рассмотрим равноускоренное движение в игре, где начальное положение объекта — `x=50` пикселей, начальная скорость — 4 пикселя за кадр, а ускорение — 2 пикселя/кадр^2. Чтобы найти положение объекта в любой момент времени, можно использовать выражение C/C++

```
x = 50 + 4*t + (2*t*t)/2;
```

где t — просто номер кадра.

В табл. 13.1 представлены несколько положений объекта для первых кадров, а на рис. 13.7 показан график зависимости скорости от времени и положения объекта на разных кадрах.

Таблица 13.1. Движение объекта с постоянным ускорением

Время/кадр (t)	*Положение (x)*	$\Delta x = x_t - x_{t-1}$
0	50	
1	$50+4*1+2*1^2/2 = 55$	5
2	$50+4*2+2*2^2/2 = 62$	7
3	$50+4*3+2*3^2/2 = 71$	9
4	$50+4*4+2*4^2/2 = 82$	11
5	$50+4*5+2*5^2/2 = 95$	13

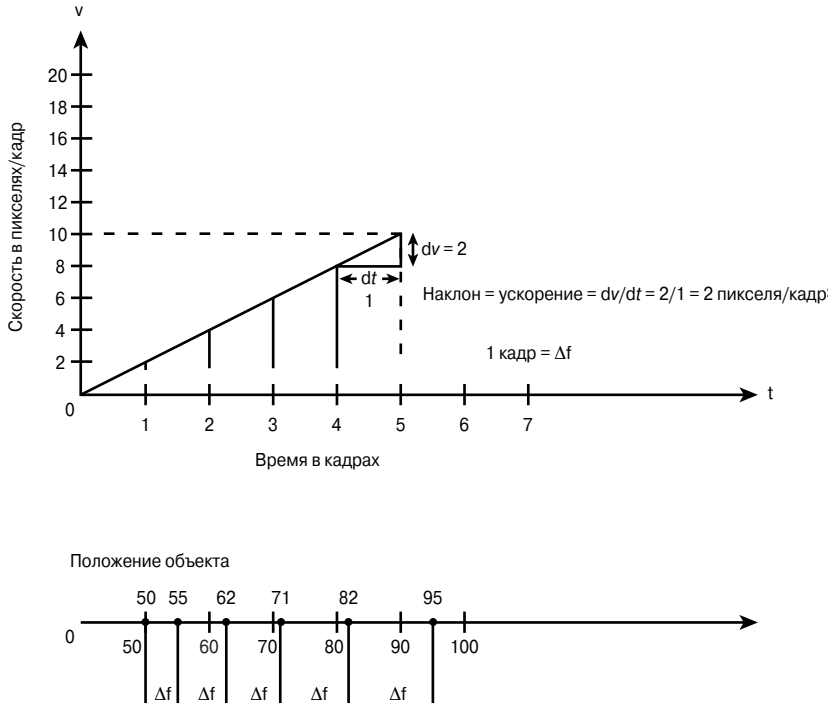


Рис. 13.7. Виртуальное ускорение в пикселях/кадр²

Моделирование ускоренного движения на C/C++ лучше всего выполнять при помощи переменной, которая хранит значение ускорения, и изменять на каждом шагу значение скорости; такой подход позволяет в будущем легко перейти к моделированию движения с переменным ускорением.

```
int acceleration = 2,  // 2 пикселя/кадр^2
   velocity     = 0,   // Начальная скорость
   x            = 0;   // Начальное положение


// В каждом цикле перемещения,
// рассматривая кадр как единицу времени
velocity += acceleration;

x += velocity;
```

НА ЗАМЕТКУ

Разумеется, этот пример для одномерного случая. Вы можете легко исправить его для работы в двухмерном случае, просто добавив координату y (а также соответствующую скорость и ускорение).

Чтобы увидеть ускоренное движение в действии, обратитесь к демонстрационной программе DEMO13_2.CPP на прилагаемом компакт-диске, в которой вы можете запустить ускоренно двигающуюся ракету. В процессе движения ракеты вы можете изменять ускорение ее движения. Обратите внимание на то, как выбор разного ускорения создает впечатление различной массы ракеты.

Сила (F)

Одной из важнейших концепций физики является *сила*. На рис. 13.8 показан один из вариантов иллюстрации силы. На нем изображен объект с массой m, находящийся на плоскости в поле тяжести Земли, ускорение свободного падения в котором обозначается как g.

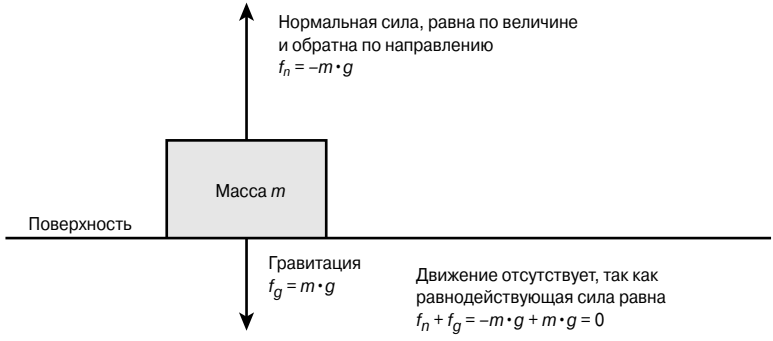


Рис. 13.8. Сила и вес

Связь между силой, массой и ускорением описывается вторым законом Ньютона: $F = ma$. Другими словами, сила, прилагаемая к объекту для его движения с ускорением a, равна произведению массы на это ускорение. Второй закон Ньютона может быть переписан в другом виде: $a = F/m$, что определяет ускорение движения тела под действием силы.

Теперь рассмотрим единицу измерения силы. Исходя из второго закона Ньютона, очевидно, что размерность силы равна $\mathrm{kg} \cdot \mathrm{m}/\mathrm{s}^2$. Для этой единицы имеется отдельное название — Ньютон (N). Так, например, представим, что объект массой 100 kg движется с ускорением $2\ \mathrm{m}/\mathrm{s}^2$. В таком случае можно утверждать, что к телу приложена сила $200\ \mathrm{kg} \cdot 2\ \mathrm{m}/\mathrm{s}^2 = 200\ \mathrm{N}$.

В видеоиграх концепция силы используется по разным причинам и для разных целей. Приведем некоторые ситуации, где необходимо использование силы.

- Вы намерены применить силу к некоторому объекту и вычислить получившееся ускорение.
- Два объекта сталкиваются, и вы хотите вычислить силы, действующие на каждый из объектов.
- Пушка в игре использует одинаковый пороховой заряд, но ядра разной массы, и вы хотите рассчитать ускорения ядер для определения дальности стрельбы.

Силы в многомерном пространстве

Конечно, силы могут действовать во всех трех измерениях, а не только вдоль прямой линии. Взглянем, например, на рис. 13.9, где изображены три силы, действующие на частицу на двухмерной плоскости. Результирующая *равнодействующая* сила представляет собой сумму всех действующих на частицу сил. Однако в данном случае суммирование следует выполнять векторно ввиду векторной природы силы. Однако векторы сил могут быть разложены на компоненты, представляющие собой проекции вектора на оси декартовой системы координат, которые могут быть суммированы отдельно и давать соответствующие компоненты равнодействующей силы.

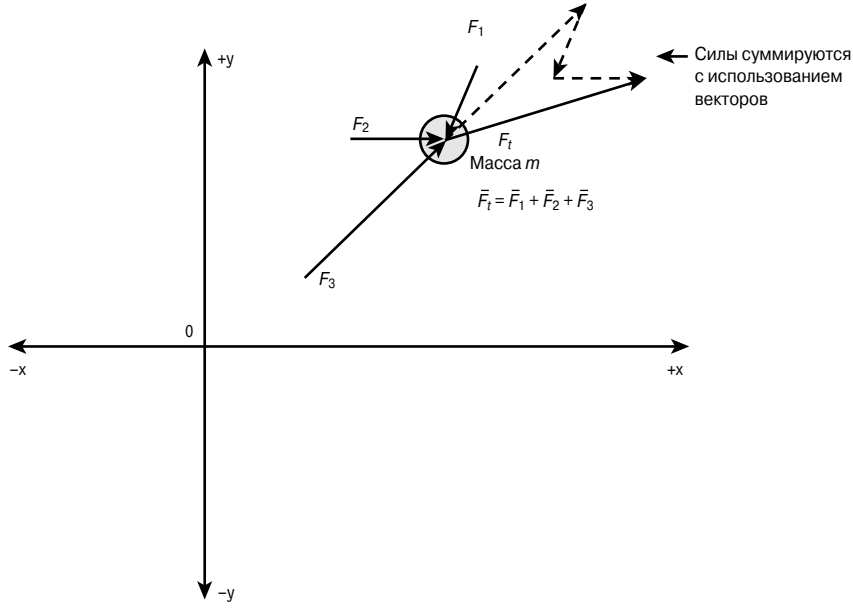


Рис. 13.9. Силы, действующие на частицу в двухмерной модели

В примере на рис. 13.9 на частицу действуют три силы — F_1, F_2 и F_3. Равнодействующая этих сил определяется как $\vec{F} = \vec{F}_1 + \vec{F}_2 + \vec{F}_3$ или

$$F_x = F_{1x} + F_{2x} + F_{3x}, \quad F_y = F_{1y} + F_{2y} + F_{3y}.$$

Это правило можно обобщить на случай любого количества сил и измерений:

$$\vec{F} = \sum_{i=1}^{n} \vec{F}_i \ .$$

Или рассматривая проекции на декартовы оси координат:

$$F_x = \sum_{i=1}^{n} F_{ix}, \quad F_y = \sum_{i=1}^{n} F_{iy}, \quad F_z = \sum_{i=1}^{n} F_{iz}.$$

Импульс (P)

Импульс — одно из тех понятий, которые трудно объяснить устно. Это свойство движущегося объекта, которое определяется как произведение массы объекта на скорость движения:

$$P = mv \ .$$

Очевидно, что единица измерения импульса — kg·m/s.

Строго второй закон Ньютона записывается в следующем виде: $\vec{F} = \frac{\mathrm{d}\vec{p}}{\mathrm{d}t}$. Для постоянной массы объекта можно преобразовать данную запись второго закона Ньютона к тому виду, с которым вы сталкивались ранее:

$$\vec{F} = \frac{\mathrm{d}\vec{p}}{\mathrm{d}t} = \frac{\mathrm{d}(m\vec{v})}{\mathrm{d}t} = m\frac{\mathrm{d}\vec{v}}{\mathrm{d}t} = m\vec{a} \ .$$

Таким образом, говоря простым языком, сила представляет собой скорость изменения импульса за единицу времени.

А теперь поговорим о законе сохранения импульса.

Законы сохранения

Познакомившись с импульсом, обсудим, что происходит при столкновении двух физических объектов. Позже эта тема рассматривается подробнее, а пока ограничимся упрощенным рассмотрением вопроса.

Помните, как в игре DOOM при стрельбе по бочкам они взрываются и разбрасывают во все стороны всю нечисть и/или другие бочки, попутно подрывая их? И что же должно происходить на самом деле при столкновении объектов между собой и со стенами?

Соударение объектов может быть как абсолютно упругим, так и не абсолютно упругим. Взгляните на рис. 13.10 — здесь изображен пример абсолютно упругого столкновения, когда скорость ударяющегося о стену объекта меняет свое направление на обратное, не меняя при этом скалярное значение, так что кинетическая энергия остается неизменной. Если же столкновение не абсолютно упругое, часть кинетической энергии переходит в тепловую и теряется. В любом случае общий импульс в системе сохраняется (правда, в данном случае может *показаться*, что это не так, но здесь следует не забывать об импульсе стены).

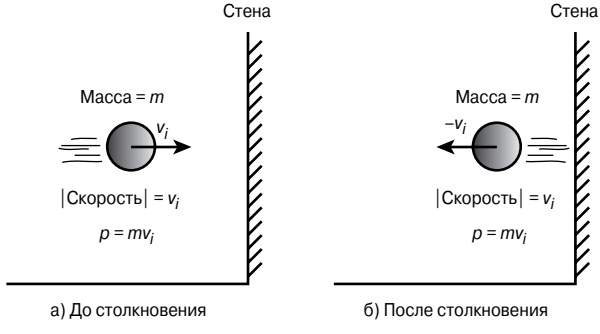


Рис. 13.10. Абсолютно упругое соударение мяча и стены

Однако мы сами творим свои миры, так почему бы не сделать их идеальными? Давайте решим задачу абсолютно упругого столкновения двух объектов в одномерном случае (к двухмерному случаю мы перейдем позже).

На рис. 13.11 показано соударение двух объектов A и B с массами m_a и m_b и скоростями v_{ai} и v_{bi} соответственно. Нужно ответить на вопрос, с какими скоростями будут двигаться эти объекты после абсолютно упругого соударения в предположении отсутствия трения (о трении речь пойдет позже).

Начнем с *закона сохранения импульса*, который гласит, что в замкнутой системе (т.е. на которую не действуют внешние силы) суммарный импульс сохраняется при любых взаимодействиях между объектами, т.е. в нашем случае справедливо уравнение $m_a v_{ai} + m_b v_{bi} = m_a v_{af} + m_b v_{bf}$.

Однако понятно, что одного уравнения для нахождения двух неизвестных величин — скоростей после соударения — явно недостаточно. И здесь нам на помощь приходит то, что соударение абсолютно упругое, т.е. по определению вся кинетическая энергия таковой и остается. Следовательно, мы можем записать *закон сохранения энергии.*

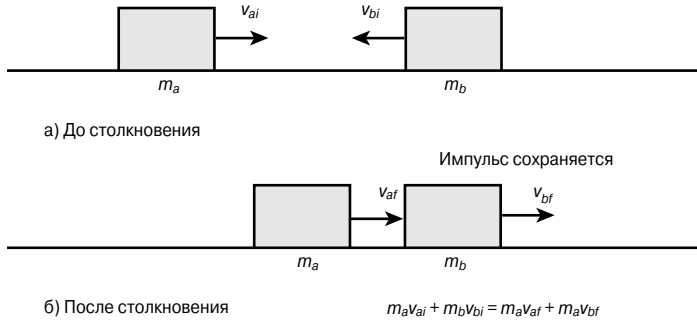


Рис. 13.11. Соударение двух объектов в одномерном случае

Кинетическая энергия в чем-то похожа на импульс, но не зависит от направления и вычисляется по формуле $K = \frac{mv^2}{2}$. Единица измерения энергии, как следует из формулы, kg·m^2/s^2. Эта единица имеет самостоятельное название — джоуль (J).

Итак, раз у нас нет потерь кинетической энергии из-за деформации объектов, потерь, связанных с трением, и прочих, закон сохранения энергии гласит:

$$\frac{m_a v_{ai}^2}{2} + \frac{m_b v_{bi}^2}{2} = \frac{m_a v_{af}^2}{2} + \frac{m_b v_{bf}^2}{2}.$$

Теперь, решая два уравнения совместно, можно легко (или с трудом, если вы плохо учились в школе) найти результат:

$$v_{af} = \frac{2m_b v_{bi} + (m_a - m_b) v_{ai}}{m_a + m_b},$$
$$v_{bf} = \frac{2m_a v_{ai} + (m_b - m_a) v_{bi}}{m_a + m_b}.$$

Так, при соударении тел с массами 2 и 3 kg, движущимися со скоростями соответственно 4 и -2 m/s, их скорости после абсолютно упругого соударения станут равны соответственно -3.2 и 2.8 m/s.

Как видите, в одномерном случае решение задачи столкновения двух тел не представляет сложности, но при двух- и трехмерном движении задача становится существенно сложнее. Но это мы обсудим несколько позже, а сейчас рассмотрим еще одну очень интересную тему.

Гравитация

Это один из наиболее распространенных эффектов, которые программистам приходится моделировать в своих играх. Гравитация — это сила, которая притягивает любой объект во Вселенной к другим объектам. Эта сила невидима и, в отличие от электромагнитного поля, не может быть экранирована.

В действительности гравитация является не силой, а следствием искривления пространства при наличии массы (рис. 13.12). Такое искривление приводит к разнице потенциальных энергий в разных точках пространства, и объект в гравитационном поле “скатывается” к другому объекту.

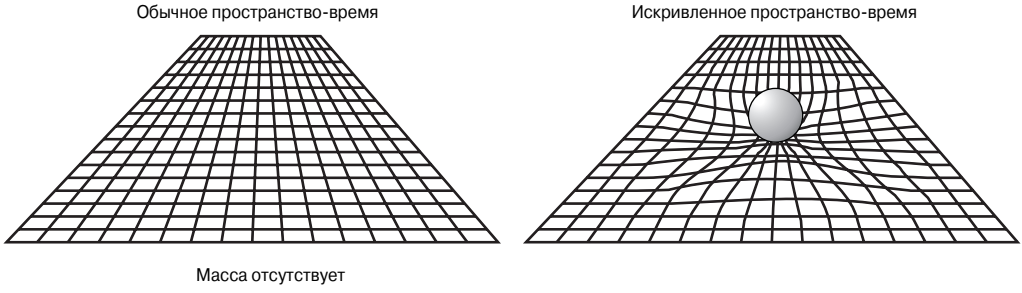


Рис. 13.12. Гравитация и пространство-время

Но нам незачем беспокоиться об эффектах общей теории относительности. Наша задача гораздо проще: смоделировать влияние гравитации на движение объектов. Для этого нужно рассмотреть два основных случая, представленных на рис. 13.13:

- когда два или большее количество объектов имеют массу одного порядка;
- когда масса одного тела на много порядков превышает массу другого тела.

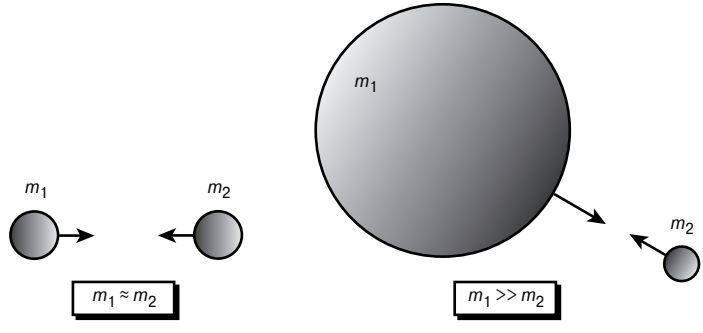


Рис. 13.13. Гравитационное взаимодействие тел

Второй случай, по сути, является частным случаем первого. В школе вам объясняли, что, если вы сбросите с крыши здания футбольный мяч или холодильник, оба они будут иметь одинаковую скорость при падении на землю. Теоретически разница есть, хотя она настолько ничтожна, что вы ее не заметите (конечно, здесь не учитываются другие силы, кроме гравитации, например сопротивление воздуха), и связана она именно с тем, что Земля также имеет конечную массу.

Поговорим теперь о математике, которая описывает гравитацию. Сила гравитационного притяжения между двумя телами с массами m_1 и m_2 равна

$$F = \gamma \frac{m_1 m_2}{r^2},$$

где γ— гравитационная постоянная, значение которой равно $6.67 \cdot 10^{-11}$ N·m^2/kg^2.

Попробуем оценить гравитационное притяжение двух людей среднего веса, находящихся на расстоянии 1 m: $F \approx 6.67 \cdot 10^{-11} \cdot 70 \cdot 70 / 1^2 \approx 3.27 \cdot 10^{-7}$ N. Очень мало? Тогда посмотрим, чему будет равна сила притяжения между человеком и Землей на расстоянии одного метра. Масса Земли составляет примерно $5.98 \cdot 10^{24}$ kg:

$F \approx 6.67 \cdot 10^{-11} \cdot 70 \cdot 5.98 \cdot 10^{24} / 1^2 \approx 2.79 \cdot 10^{16}$ N. Понятно, что такая сила просто расплющит человека, но ошибка в том, что здесь мы считали, что вся масса Земли сосредоточена в шаре размером 1 m. Но поскольку радиус Земли составляет примерно $6.38 \cdot 10^6$ m, то и сила, действующая на человека на поверхности Земли, будет составлять $F \approx 6.67 \cdot 10^{-11} \cdot 70 \cdot 5.98 \cdot 10^{24} / (6.38 \cdot 10^6)^2 \approx 686$ N. Вполне разумное число.

При вычислениях шар массой m и радиусом r можно заменять точечной массой той же величины, расположенной в центре шара, если шар изотропен (т.е. его плотность во всех направлениях одинакова), а расстояние до интересующих нас объектов от центра шара превышает r.

Итак, теперь вы знаете, как вычислить силу гравитационного притяжения между двумя объектами, и можете использовать эту простейшую модель в игре. Конечно, вы не обязаны использовать значение гравитационной постоянной, равное $6.67 \cdot 10^{-11}$ N·m^2/kg^2, — ведь вы сами творите свой мир и можете делать эту постоянную какой угодно. Важно только, чтобы гравитационное притяжение было пропорционально массам объектов и обратно пропорционально квадрату расстояния между ними.

Моделирование гравитации

Используя рассмотренную формулу гравитационного притяжения, вы можете моделировать, например, черную дыру в космической игре (впрочем, если это действительно черная дыра, то без общей теории относительности тут не обойтись). Для этого вы просто выбираете подходящую для вашего мира гравитационную постоянную и приступаете к работе. На каждом шаге вы вычисляете действующую на космический корабль силу, а значит, и его ускорение, а также новую скорость и местоположение корабля.

На прилагаемом компакт-диске находится демонстрационная программа `DEMO13_3.CPP`, в которой выполняется моделирование движения космического корабля в гравитационном поле. Вы можете управлять движением корабля по экрану, но берегитесь черной дыры!

Еще одно использование гравитации в играх — корректное моделирование падения объектов с неба (здания, вышки и т.д.). Здесь ситуация существенно упрощается, так как, с одной стороны, масса падающего объекта на много порядков меньше массы притягивающего тела (например, Земли), а проходимые при этом телом расстояния гораздо меньше расстояния до центра притягивающего тела. Поэтому с достаточно большой точностью на протяжении всего пути падающего тела силу гравитационного взаимодействия можно считать постоянной (рис. 13.14), чего, конечно же, нельзя было делать в предыдущем примере с черной дырой.

В такой ситуации можно сделать ряд предположений, которые значительно упрощают решение задачи. Первое состоит в том, что ускорение, вызванное гравитационным притяжением, постоянно для всех падающих тел, независимо от их массы, и равно примерно 9.8 m/s^2. Этот вывод позволяет использовать уже рассмотренную ранее модель равноускоренного движения:

$$v(t) = v_0 + gt,$$
$$y(t) = y_0 + v_0 t + \frac{gt^2}{2},$$

где g — только что рассмотренное ускорение свободного падения.

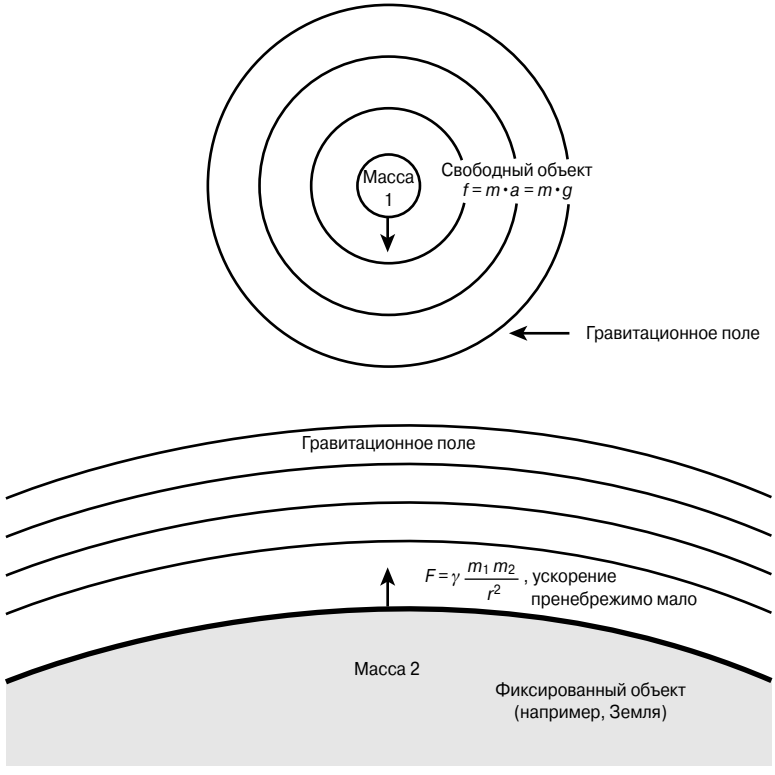


Рис. 13.14. Гравитационное притяжение

При падении объекта без начальной скорости из точки с координатой $y_0 = 0$ формула упрощается до $y(t) = \frac{gt^2}{2}$. Естественно, в реальной игре конкретное значение *g* выбирается таким, которое наиболее подходит для создаваемого виртуального мира, а время просто представляет собой номер кадра. При этом падение мяча с вершины экрана можно смоделировать, например, таким образом:

```
int y_pos     = 0,   // Верх экрана
   y_velocity = 0,  // Начальная скорость
   gravity    = 1;    // Ускорение свободного падения

// Выполняем цикл падения до тех пор, пока объект не
// пересечет нижней границы экрана (SCREEN_BOTTOM)
while(y_pos < SCREEN_BOTTOM)
{
   // Обновляем положение объекта
   y_pos += y_velocity;
   // Обновляем скорость объекта
   y_velocity += gravity;
} // while
```

СОВЕТ

Я использовал для вычисления положения объекта скорость, а не непосредственную формулу — так проще проводить вычисления.

Вы можете спросить, а как сделать, чтобы объект падал по криволинейной траектории? Это легко: нужно просто дополнительно перемещать объект по оси X с постоянной скоростью, и его движение будет выглядеть как движение брошенного, а не падающего объекта.

```
int y_pos     = 0,   // Верх экрана
   y_velocity = 0,   // Начальная скорость по горизонтали
   x_velocity = 2,   // Начальная скорость по вертикали
   gravity    = 1;   // Ускорение свободного падения

// Выполняем цикл падения до тех пор, пока объект не
// пересечет нижней границы экрана (SCREEN_BOTTOM)
while(y_pos < SCREEN_BOTTOM)
{
   // Обновляем положение объекта
   x_pos += x_velocity;
   y_pos += y_velocity;
   // Обновляем скорость объекта
   y_velocity += gravity;
} // while
```

Моделирование траекторий снарядов

Падающие объекты вполне увлекательны, но в играх есть еще более увлекательные моменты, например стрельба из пушки. Вы уже можете самостоятельно смоделировать движение снаряда? В принципе все, что вы должны для этого знать, вы уже знаете. Если не можете, то этот раздел поможет вам. Взгляните на рис. 13.15, где задача представлена в графическом виде. У нас имеется плоскость $y = 0$, танк с координатами $(0, 0)$ с пушкой, направленной под углом θ к оси X. Что произойдет, если выстрелить снарядом массы m со скоростью v_i? Как далеко улетит снаряд? Сопротивление воздуха отсутствует.

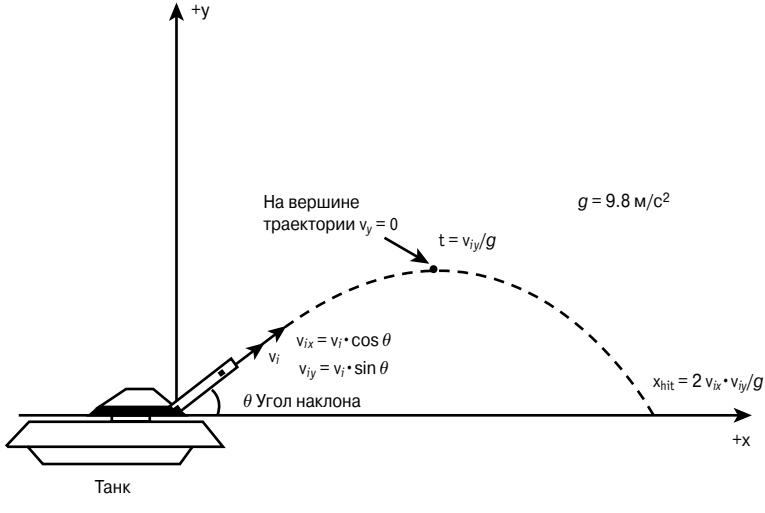


Рис. 13.15. Определение траектории снаряда

Эту задачу можно решить, рассматривая компоненты движения по осям X и Y в отдельности. Сначала разложим начальную скорость v_i по осям:

$$v_{ix} = v_i \cos\theta,$$
$$v_{iy} = v_i \sin\theta.$$

Теперь на минуту забудем о части, связанной с движением по горизонтали, и рассмотрим движение по вертикали. При движении только по вертикали снаряд сначала поднимется вверх, а затем упадет вниз. Его движение описывается следующими уравнениями:

$$v_y(t) = v_{iy} - gt,$$
$$y(t) = v_{iy} - \frac{gt^2}{2}.$$

Здесь нет члена y_0, поскольку начальные координаты снаряда, как уже отмечалось, равны $(0,0)$, а знак "минус" перед g указывает на то, что ускорение свободного падения направлено противоположно направлению оси Y выбранной нами системы координат.

Известно, что время, которое требуется для достижения вершины траектории, равно времени, необходимому для падения снаряда на землю из этой верхней точки. Это время можно определить исходя из того, что в верхней точке траектории вертикальная скорость снаряда равна 0, так что $t = v_{iy}/g = v_i \sin\theta/g$. Очевидно, что общее время полета снаряда равно $2t$. Теперь самое время вновь обратиться к движению снаряда вдоль оси X, которое происходит с одной и той же скоростью v_{ix}. Следовательно, полное расстояние, которое пройдет снаряд до падения на землю, рассчитывается как его скорость v_{ix}, умноженная на время полета, равное $2v_{iy}/g$. Итак, дальность стрельбы равна

$$x_{hit} = \frac{2v_{ix}v_{iy}}{g} = \frac{2v_i \cos\theta\, v_i \sin\theta}{g} = \frac{v_i^2}{g}\sin 2\theta\ .$$

Итак, мы получили ответ на поставленный в задаче вопрос. Не менее легко найти, например, максимальную высоту подъема снаряда, но сейчас нас интересует, как же смоделировать движение снаряда на экране? Все очень просто: как и ранее, моделируется движение снаряда по осям X и Y в соответствии с обычными формулами кинематики. Следует только учесть, что на экране ось Y направлена противоположно оси Y на рис. 13.15, да и танк располагается по оси Y не в нулевой точке, а в нижней точке экрана. Вот соответствующий фрагмент кода:

```
float x_pos     = 0,                  // Начальные координаты
    y_pos     = SCREEN_BOTTOM,  // снаряда
    x_velocity = 0,                  // Скорость снаряда
    y_velocity = 0,                  // по осям X и Y
    gravity   = 1,                   // Ускорение
    velocity  = INITIAL_VEL,         // Начальная скорость
    angle     = INITIAL_ANGLE;    // Угол стрельбы

// Вычисляем начальные скорости вдоль осей координат
x_velocity =  velocity * cos(angle);
y_velocity = -velocity * sibn(angle);

// Выполняем вычисление траектории до тех пор, пока
// снаряд не пересечет нижней границы экрана
while(y_pos <= SCREEN_BOTTOM)
{
  // Обновляем координаты снаряда
  x_pos += x_velocity;
```

```
    y_pos += y_velocity;

    // Обновляем скорость снаряда
    y_velocity += gravity;
} // while
```

И это все. Если вы захотите добавить сопротивление воздуха, то в первом приближении его можно смоделировать как небольшое ускорение, направленное против движения по оси X:

```
x_velocity -= wind_factor;
```

Значение переменной wind_factor должно быть очень мало, например порядка 0.01 (иначе возможна ситуация, когда снаряд просто полетит назад).

На прилагаемом компакт-диске находится соответствующая демонстрационная программа DEMO13_4.CPP, копия экрана которой показана на рис. 13.16. В этой программе вы можете управлять стрельбой из пушки и смотреть, как влияют на траекторию угол прицеливания, сила притяжения и ветра.

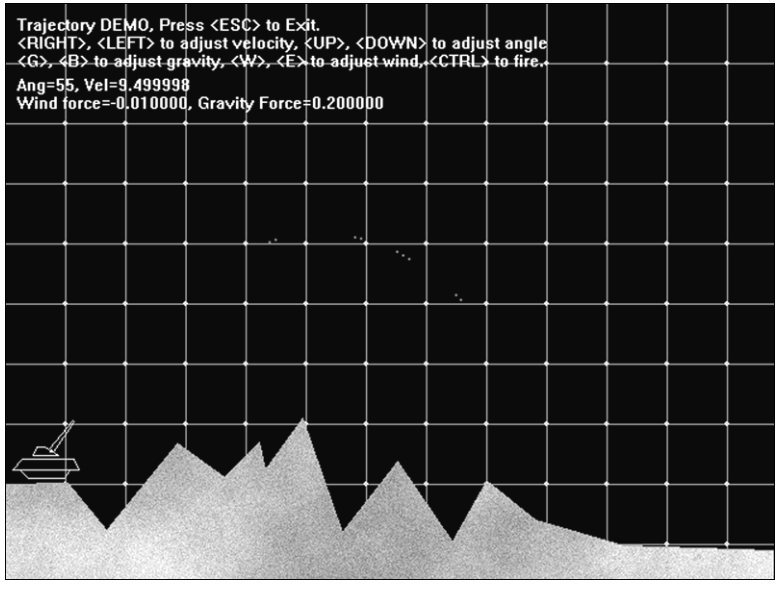


Рис. 13.16. Работа программы DEMO13_4.EXE

Трение

Следующий вопрос, который я хочу рассмотреть, — это трение. Трением, по сути, является любая сила, которая замедляет движение или поглощает энергию из другой системы. Например, в автомобиле 30–40% вырабатываемой двигателем энергии уходит в тепло или затрачивается на механическое трение. С этой точки зрения самым эффективным средством передвижения, пожалуй, является велосипед, потери которого составляют не более 10–20%.

Основы трения

Трение представляет собой сопротивление в направлении, противоположном направлению движения и, следовательно, может быть смоделировано при помощи использова-

ния силы, которую обычно называют силой трения. Взгляните на рис. 13.17, где изображена стандартная модель трения тела с массой m на плоскости.

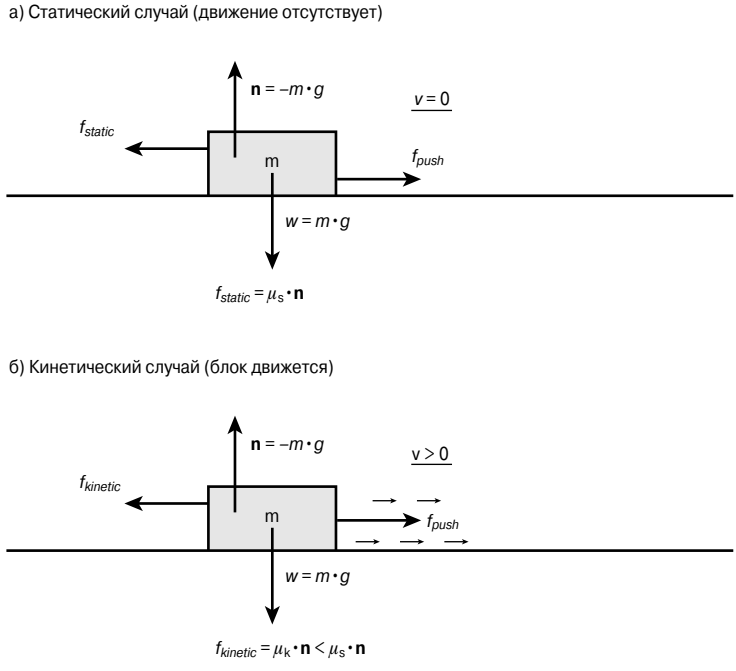


Рис. 13.17. Модель трения

Если вы попытаетесь толкать тело параллельно плоскости, на которой оно расположено, то столкнетесь с сопротивлением, силу которого математически можно записать как

$$F_{f_{static}} = mg\mu_s ,$$

где m — масса объекта, g — ускорение свободного падения, а μ_s — статический коэффициент трения, который зависит от различных условий и материала объекта и плоскости. Если сила F, которую вы прилагаете к объекту, превышает F_f, то объект начинает двигаться. Обычно при движении коэффициент трения несколько уменьшается и становится равным кинетическому коэффициенту трения μ_k:

$$F_{f_{kinetic}} = mg\mu_k .$$

Когда вы перестаете прикладывать силу, объект под действием силы трения замедляет движение и в конечном итоге останавливается.

Все, что вам требуется для моделирования трения на плоской поверхности, — это уменьшение скорости всех движущихся объектов до тех пор, пока объект не остановится. Вот пример фрагмента кода, в котором смоделировано замедление движения объекта с первоначальной скоростью 16 пикселей за кадр под действием трения:

```
int x_pos = 0,        // Начальное положение
  x_velocity = 16,  // Начальная скорость
  friction = -1;      // Трение

// Перемещаем объект, пока его скорость не станет <= 0
```

```
while(x_velocity > 0)
{
  // Перемещение объекта
  x_pos += x_velocity;
  // Действие трения
  x_velocity += friction;
} // while
```

Первое, на что вы должны обратить внимание, — это сходство данной модели с моделью гравитации. Эти модели почти идентичны, что не случайно, так как обе модели описываются одними и теми же законами Ньютона в системе с постоянной силой. Поэтому данная модель применима для имитации движения объекта под действием любых сил. Кроме того, к объекту может быть приложено несколько сил — в этом случае нужно просто их суммировать.

В качестве демонстрационного приложения, моделирующего действие силы трения, рассмотрите программу DEMO13_5.CPP на прилагаемом компакт-диске. Копия экрана данной программы показана на рис. 13.18. Программа позволяет вам запустить в произвольном направлении хоккейную шайбу на виртуальном поле, свойства которого (силу трения) вы можете изменять. Шайба постепенно замедляет движение под действием силы трения.

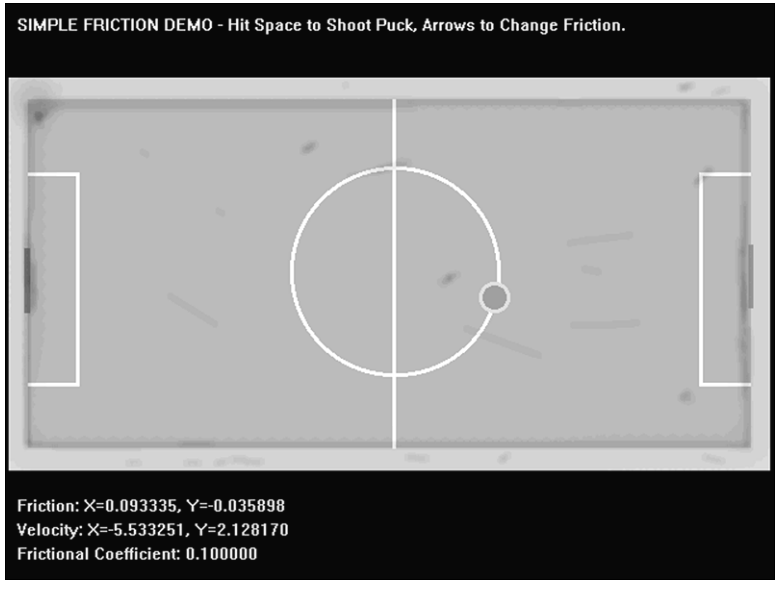


Рис. 13.18. Демонстрация действия сил трения

Трение на наклонной плоскости

Главный вывод, который можно сделать из рассмотренного материала, состоит в том, что трение можно смоделировать как простую силу сопротивления, приводящую к уменьшению скорости в каждом цикле игры. Но я бы не хотел на этом останавливаться и в качестве примера анализа более сложной задачи хочу рассмотреть движение тела по наклонной плоскости с учетом силы трения. Должен сразу предупредить вас — здесь будет много работы с векторами, поэтому, если вы не уверены в себе, рекомендую вернуться к главе 8, “Растеризация векторов и двухмерные преобразования”, где изложен материал о векторах.

На рис. 13.19 показана задача, которую мы будем решать: движение объекта массой m по наклонной плоскости, угол наклона которой к горизонтали равен θ. Статический и кинетический коэффициенты трения равны соответственно μ_s и μ_k. Начнем с описания объекта в положении равновесия, т.е. неподвижного объекта. В этом случае сумма всех сил как вдоль оси X, так и вдоль оси Y должна быть равна 0.

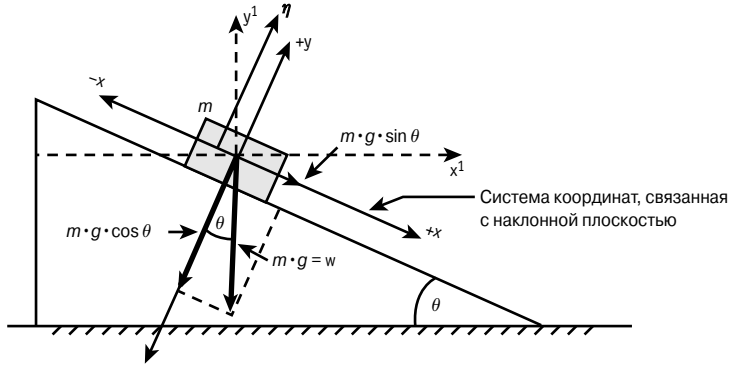


Рис. 13.19. Движение тела но наклонной плоскости

Однако сначала познакомимся с концепцией *нормальной силы*. Это не что иное, как сила, с которой плоскость поддерживает объект, чтобы он не мог двигаться в направлении, перпендикулярном плоскости. Очевидно, что эта сила всегда направлена перпендикулярно плоскости. Так, когда вы стоите на горизонтальной плоскости, эта сила направлена вертикально вверх и численно равна весу, с которым вы действуете на плоскость. Обозначим вектор нормальной силы как $\boldsymbol{\eta}$.

Теперь перейдем в систему координат, связанную с наклонной плоскостью. В этой системе координат ось X направлена вдоль плоскости вниз, а ось Y — перпендикулярно плоскости. Условие равновесия можно записать следующим образом: $\sum \mathbf{F} = 0$, т.е. сумма всех сил, действующих на объект, равна 0. Какие же это силы? Это сила тяжести, нормальная сила, действующая на тело со стороны плоскости, и сила трения, направленная вдоль плоскости в сторону, противоположную возможному направлению соскальзывания тела.

Разложив эти силы по осям координат (см. рис. 13.19), получим следующие уравнения:

$$\sum F_x = mg\sin\theta - \eta\mu_s = 0,$$
$$\sum F_y = \eta - mg\cos\theta = 0.$$

Как видите, из второго уравнения следует, что значение нормальной силы меньше полного веса (mg) тела, что очевидно, поскольку нормальная сила направлена перпендикулярно к плоскости, в то время как вес — нет.

Решая приведенную систему уравнений и подставляя найденное из второго уравнения значение нормальной силы в первое, получим, что $\mu_s = \operatorname{tg}\theta$. Этот же результат можно записать в виде $\theta_{critical} = \operatorname{arctg}\mu_s$.

А теперь читайте внимательно. Нами получен следующий результат: имеется угол, называемый критическим углом наклона, при превышении которого тело на наклонной плоскости начнет скользить. Если угол наклона плоскости меньше критического, тело будет покоиться. Как видите, критический угол и коэффициент статического трения взаимосвязаны, так что для определения неизвестного нам коэффициента трения можно

разместить тело на плоскости и наклонять ее до тех пор, пока тело не начнет скользить. Тангенс угла наклона плоскости в момент начала скольжения тела и даст искомый коэффициент трения.

После того как начнется скольжение тела, сумма сил вдоль оси Y остается равной нулю, но сумма сил вдоль оси X оказывается ненулевой:

$$\sum F_x = mg\sin\theta - \eta\mu_s = mg\left(\sin\theta - \mu_s\cos\theta\right).$$

Есть одно замечание: поскольку тело приходит в движение, следует использовать коэффициент кинетического трения, т.е. тело по наклонной плоскости движется под действием силы

$$F_x = mg\left(\sin\theta - \mu_k\cos\theta\right).$$

СЕКРЕТ

В играх точность моделирования волнует нас не настолько, чтобы нельзя было усреднить значения коэффициентов статического и кинетического трения и не использовать одну константу вместо двух.

Вычислить ускорение движения тела по наклонной плоскости очень просто: достаточно поделить найденную равнодействующую силу на массу тела:

$$a = g\left(\sin\theta - \mu_k\cos\theta\right).$$

Теперь можно легко моделировать скольжение объекта по наклонной плоскости, увеличивая скорость объекта в направлении оси X, связанной с плоскостью, с учетом полученного значения ускорения *a*. Но минутку! Ведь мы-то пишем программу для другой системы координат! Что же делать?

Ответ очень прост: разложить вектор ускорения **a**, полученного нами, на составляющие вдоль осей X и Y нашей так называемой лабораторной системы координат. Поскольку угол θ нам известен, можно указать компоненты ускорения:

$$a_x = a\cos\theta = g\left(\sin\theta\cos\theta - \mu_k\cos^2\theta\right),$$
$$a_y = -a\sin\theta = g\left(-\sin^2\theta + \mu_k\sin\theta\cos\theta\right).$$

Знак "-" у Y-компоненты появляется постольку, поскольку мы знаем, что вектор ускорения направлен вниз по оси Y. Вот фрагмент кода, который использует приведенные соотношения для моделирования движения тела по наклонной плоскости:

```
float x_pos     = SX,           // Начальное положение
    y_pos     = SY,             // объекта на плоскости
    x_velocity = 0,             // Начальная скорость
    y_velocity = 0,             // объекта
    x_plane    = 0,             // Вектор, направленный
    y_plane    = 0,             // вдоль плоскости
    gravity    = 1,             // Гравитация
    angle      = PLANE_ANGLE,   // Угол наклона (должен
                                // превышать критический)
    frictionk  = 0.1;           // Коэффициент трения

// Вычисляем вектор вдоль плоскости
x_plane = cos(angle);
y_plane = sin(angle); // Без знака "-", так как ось Y
                      // направлена вниз
```

```
// Циклическое перемещение объекта до низа экрана
while(y_pos < SCREEN_BOTTOM)
{
  // Обновление положения объекта
  x_pos += x_velocity;
  y_pos += y_velocity;

  // Обновление скорости объекта[2]
  x_velocity += x_plane*gravity*(sin(angle)-frictionk*cos(angle));
  y_velocity += y_plane*gravity*(sin(angle)-frictionk*cos(angle));

} // while
```

Главный вывод, который можно сделать из рассмотренных примеров моделирования, таков: нужно знать лежащие в основе этих моделей физические законы и тогда создание моделей становится не такой уж сложной задачей.

Столкновения

Как уже отмечалось, существует два типа столкновений: упругие и неупругие. При упругих столкновениях выполняются законы сохранения импульса и кинетической энергии; при неупругих — часть кинетической энергии переходит в тепловую.

В большинстве игр неупругие столкновения не рассматриваются в силу сложности их моделирования; более того, зачастую используемые для моделирования столкновений средства не имеют ничего общего с физикой соударений.

Простой удар

Взгляните на рис. 13.20. На нем изображена очень часто встречающаяся в играх ситуация: столкновение объекта с границами экрана. Объект имеет некоторую начальную скорость (v_x, v_y) и при движении может ударяться о любую из четырех границ экрана. Когда объект сталкивается с другим объектом, масса которого намного превышает его собственную, задача намного упрощается, поскольку мы просто считаем, что объект с гораздо большей массой остается неподвижен, и должны рассмотреть движение только одного объекта. Хорошим примером такой ситуации может служить бильярдный стол, масса которого во много раз превышает массу шаров.

Когда мяч ударяется об одну из сторон, он отражается от этой стороны под тем же углом, под которым он двигался до столкновения, но в противоположном направлении, как показано на рис. 13.20. Таким образом, для моделирования движения мяча, например по бильярдному столу или при столкновении с массивным телом, необходимо лишь вычислить направление вектора нормали в точке соударения и выполнить отражение объекта, как показано на рис. 13.21.

[2] Очевидно, что для повышения эффективности вычислений, как минимум, имеет смысл вычислить значение выражения gravity*(sin(angle)-frictionk*cos(angle)) заранее, до входа в цикл, и использовать в цикле сохраненное значение, не вычисляя его заново. — *Прим. ред.*

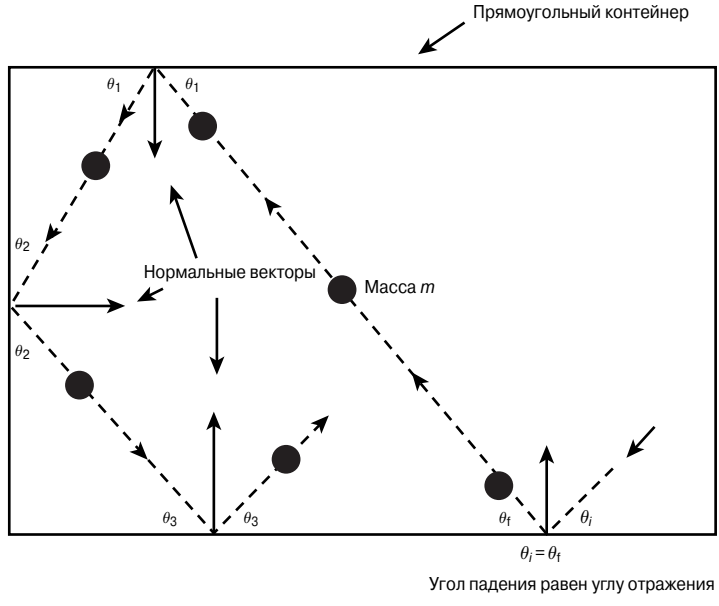


Рис. 13.20. Ударяющийся мяч

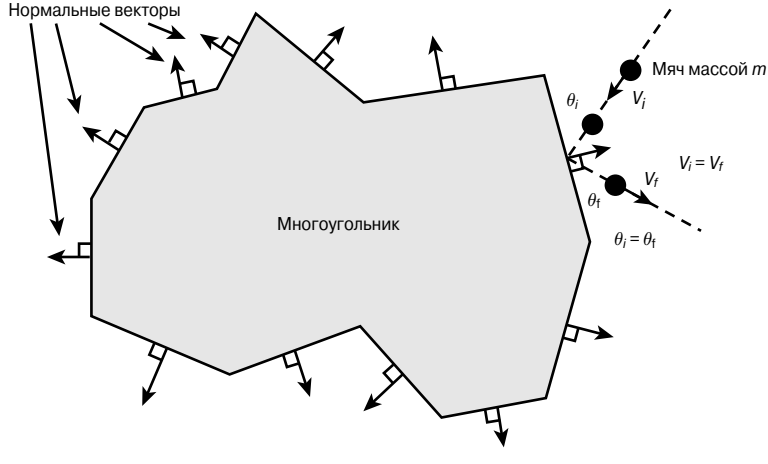


Рис. 13.21. Столкновение мяча с объектом с плоскими гранями

Хотя это не так сложно описать словами, реализация потребует массу тригонометрических вычислений. Однако возможен и более простой путь, нужно только хорошо разобраться в физике происходящего процесса. Достаточно абстрагироваться от углов и взглянуть на задачу "физическим" взглядом, как станет ясно: вместо того чтобы иметь дело с углами, следует подумать о скоростях. Если мяч ударяется о вертикальные границы, то его вертикальная скорость в результате столкновения остается прежней, а горизонтальная меняет знак. Если же столкновение происходит с горизонтальной границей, то знак меняется у вертикальной скорости, а горизонтальная остается той же, что и до удара:

```
// Проверка соударения с вертикальной стеной
if (x >= EAST_EDGE || x <= WEST_EDGE)
   vx = -vx; // Изменяем горизонтальную скорость

// Проверка соударения с горизонтальной стеной
if (y >= SOUTH_EDGE || y <= NORTH_EDGE)
   vy = -vy; // Изменяем вертикальную скорость
```

Конечно, такое упрощение работает только при столкновениях с горизонтальными и вертикальными барьерами. При столкновении со стенами, которые не являются параллельными оси X или Y, требуются более сложные вычисления углов.

СЕКРЕТ

Если вы хотите использовать описанный способ в качестве грубого, но быстро работающего приближения при моделировании столкновения объектов, каждый объект можно рассматривать с точки зрения другого объекта как описанный прямоугольник (рис. 13.22).

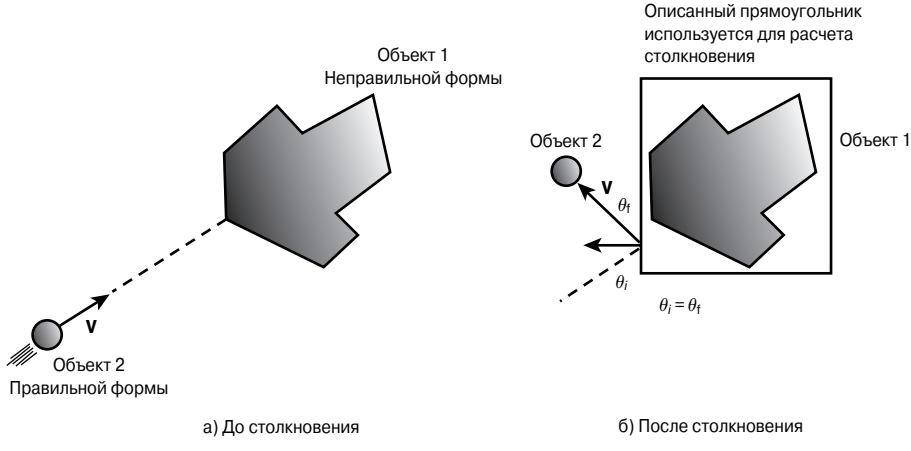


Рис. 13.22. Упрощенное столкновение объектов

Пример использования описанной технологии вы можете найти в демонстрационной программе DEMO13_6.CPP на прилагаемом компакт-диске, которая моделирует движение шаров на бильярдном столе (рис. 13.23). Заметим, что в этой программе шары друг с другом не сталкиваются, сталкиваясь только со стенками стола.

Столкновение с плоскостью произвольной ориентации

Использование прямоугольника в качестве формы объекта для разрешения проблемы столкновений неплохо работает в каком-нибудь пинг-понге, но на дворе 21 век, и решать такие вопросы нужно по-настоящему. Для этого требуется лишь вывести уравнения отражения вектора от плоской поверхности (рис. 13.24, *а*). Первое приближение, которое используется при решении задачи — что объект очень мал, а соударение абсолютно упругое. Ранее мы уже пришли к выводу, что в таком случае угол, под которым объект ударяется о стену, равен углу, под которым он отскакивает от стены. Эти углы (по отношению к вектору *нормали* к плоскости) называются соответственно углами *падения* и *отражения*. А теперь обратимся к математике. Все, что нам потребуется, — немного знаний о работе с векторами.

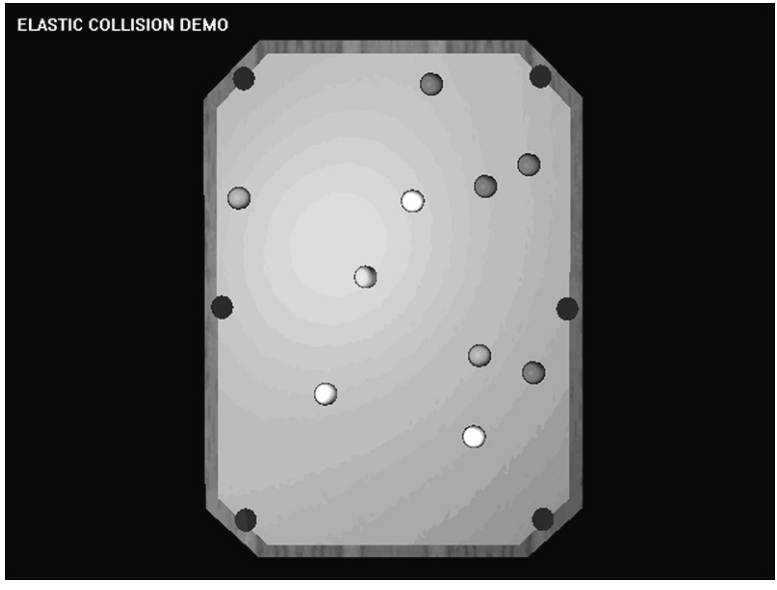


Рис. 13.23. Простая модель столкновения шаров со стенками

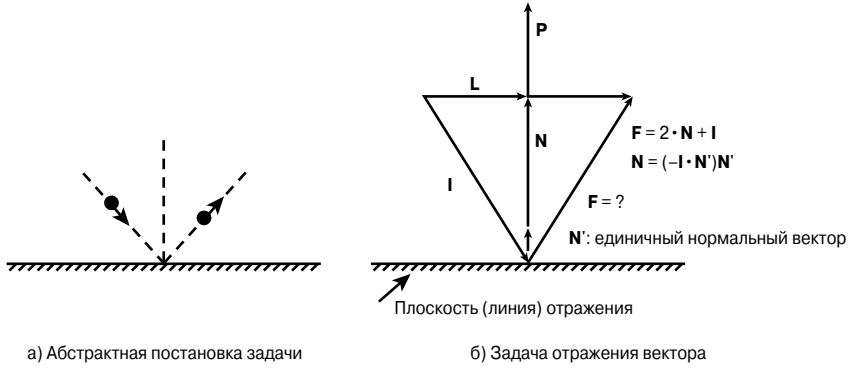


Рис. 13.24. Задача отражения вектора

На рис. 13.24, *б* введен ряд обозначений, которые помогут решить поставленную задачу. Обратите внимание, что здесь нет осей координат, так как задача решается в общем виде с использованием векторов.

Итак, вот постановка задачи. Задан вектор падения на плоскость **I** и нормальный вектор плоскости **N′**. Требуется найти вектор отражения **F**.

Сначала рассмотрим, что же такое нормальный вектор. Это нормализованный вектор **P**; но что такое вектор **P**? Это вектор, перпендикулярный плоскости (или линии в двухмерном случае), о которую ударяется мяч. Определить этот вектор можно различными способами. Он может быть вычислен предварительно и храниться в структуре данных, представляющей плоскость, а может вычисляться "на лету".

Способы вычисления вектора **P** зависят от того, как именно представлена "стена". Если стена является плоскостью в трехмерном пространстве, то мы просто находим **P** и **N′**, исходя из аналитического представления плоскости в пространстве:

$$n_x(x-x_0)+n_y(y-y_0)+n_z(z-z_0)=0,$$
$$\mathbf{P}=\langle n_x,n_y,n_z\rangle,$$
$$\mathbf{N}'=\langle n_x,n_y,n_z\rangle\Big/|\mathbf{P}|=\langle n_x,n_y,n_z\rangle\Big/\sqrt{n_x^2+n_y^2+n_z^2}\,.$$

Если же столкновение происходит в двухмерном случае и объект сталкивается с прямой линией (или отрезком), заданной двумя точками (рис. 13.25), то вычислить нормаль можно следующим образом.

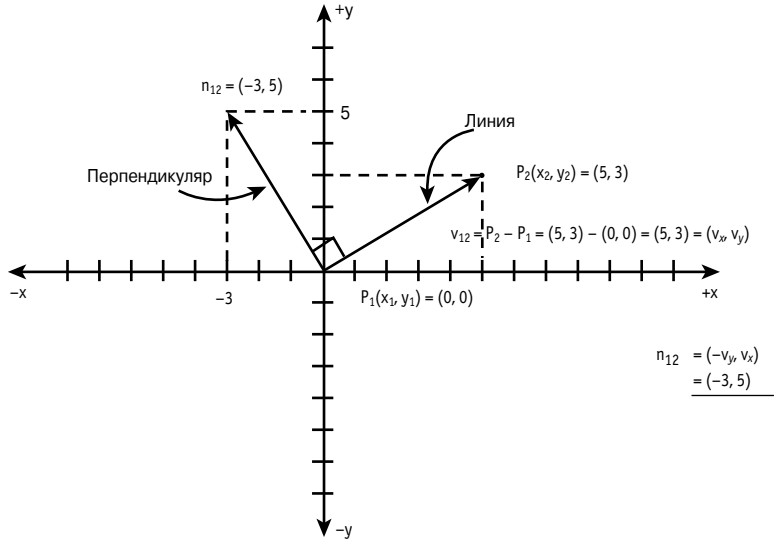


Рис. 13.25. Вычисление перпендикуляра к прямой линии

Если линия определяется двумя точками $p_1(x_1,y_1)$ и $p_2(x_2,y_2)$: $\mathbf{v}_{12}=\langle x_2-x_1,y_2-y_1\rangle=\langle v_x,v_y\rangle$, то вектор нормали к ней можно найти как $\mathbf{P}_{12}=\langle n_x,n_y\rangle=\langle -v_y,v_x\rangle$. Чтобы убедиться в этом, вспомним о скалярном произведении векторов, которое равно 0, если векторы взаимно перпендикулярны. В нашем случае

$$\mathbf{v}_{12}\cdot\mathbf{P}_{12}=\langle v_x,v_y\rangle\cdot\langle -v_y,v_x\rangle=-v_xv_y+v_yv_x=0\,.$$

Чтобы получить вектор $\mathbf{N}'$, нужно просто нормализовать вектор $\mathbf{P}$, для чего достаточно разделить его на собственную длину:

$$\mathbf{N}'=\mathbf{P}\Big/|\mathbf{P}|=\langle -v_y,v_x\rangle\Big/\sqrt{(-v_y)^2+(v_x)^2}\,.$$

Вернемся, однако, к нашему выводу. Итак, к этому моменту у нас имеется нормальный вектор $\mathbf{N}'$, который не следует путать с вектором $\mathbf{N}$, представляющим собой *проекцию вектора* $\mathbf{I}$ *на вектор* $\mathbf{N}'$. Звучит страшно? Но на самом деле ничего страшного в этом нет. Представьте себе солнечный свет, падающий перпендикулярно на вектор $\mathbf{N}'$ (на рис. 13.24, *б* слева направо). Тогда проекция — это просто тень от вектора $\mathbf{I}$ на оси $\mathbf{N}'$. Эта проекция и есть интересующий нас вектор $\mathbf{N}$, зная который, нетрудно вычислить искомый вектор $\mathbf{F}$. Начнем с вычисления $\mathbf{N}$:

$$\mathbf{N}=(-\mathbf{I}\cdot\mathbf{N}')\mathbf{N}'\,,$$

т.е. **N** представляет собой скалярное произведение векторов -**I** и **N′**, умноженное на вектор **N′**.

Теперь еще раз взгляните на рис. 13.24,*б* — из него совершенно очевидны два равенства:

$$\mathbf{L} = \mathbf{N} + \mathbf{I},$$
$$\mathbf{F} = \mathbf{N} + \mathbf{L},$$

откуда легко выводится требующееся нам соотношение $\mathbf{F} = 2\mathbf{N} + \mathbf{I}$.

Пример отражения вектора

Мне почему-то кажется, что одних теоретических выкладок для полного понимания решения задачи мало, так что давайте рассмотрим конкретный пример, показанный на рис. 13.26.

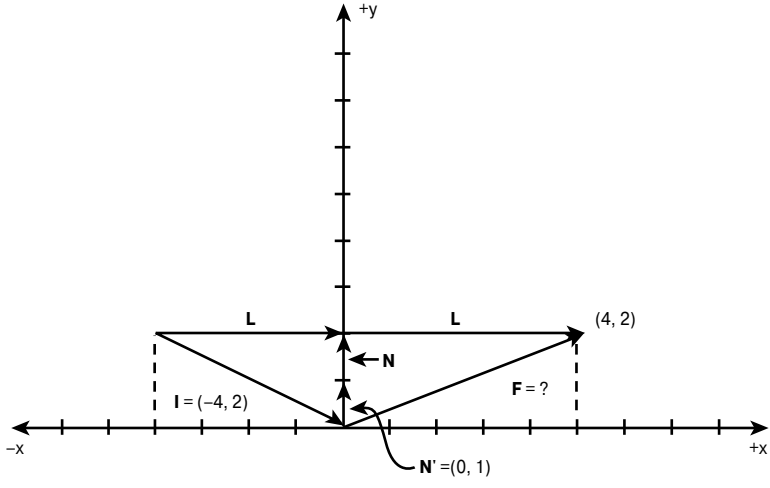


Рис. 13.26. Конкретный пример отражения

Итак, мы имеем $\mathbf{I} = \langle 4,2 \rangle$, $\mathbf{N}' = \langle 0,1 \rangle$, так что простые выкладки дают нам требующийся результат.

$$\begin{aligned} F &= 2\mathbf{N} + \mathbf{I} \\ &= 2(-\mathbf{I} \cdot \mathbf{N}')\mathbf{N}' + \mathbf{I} \\ &= -2(\langle 4,-2 \rangle \cdot \langle 0,1 \rangle)\langle 0,1 \rangle + \langle 4,-2 \rangle \\ &= -2(4 \cdot 0 - 2 \cdot 1)\langle 0,1 \rangle + \langle 4,-2 \rangle \\ &= \langle 0,4 \rangle + \langle 4,-2 \rangle = \langle 4,2 \rangle \end{aligned}$$

Как видите, получен тот же ответ, что и показанный на рис. 13.26. Вот и все, данную задачу вы научились решать, осталось научиться решать еще одну маленькую задачу: определять, в каком именно месте произойдет столкновение мяча со стеной.

Пересечение отрезков

Рассмотрим теперь задачу пересечения отрезков. В основном она базируется на вычислении пересечений линий, однако нас интересуют не линии, а именно отрезки. Прямая линия продолжается в обе стороны неограниченно, в то время как отрезок ограничен (рис. 13.27); именно в этом и кроется главное различие этих задач.

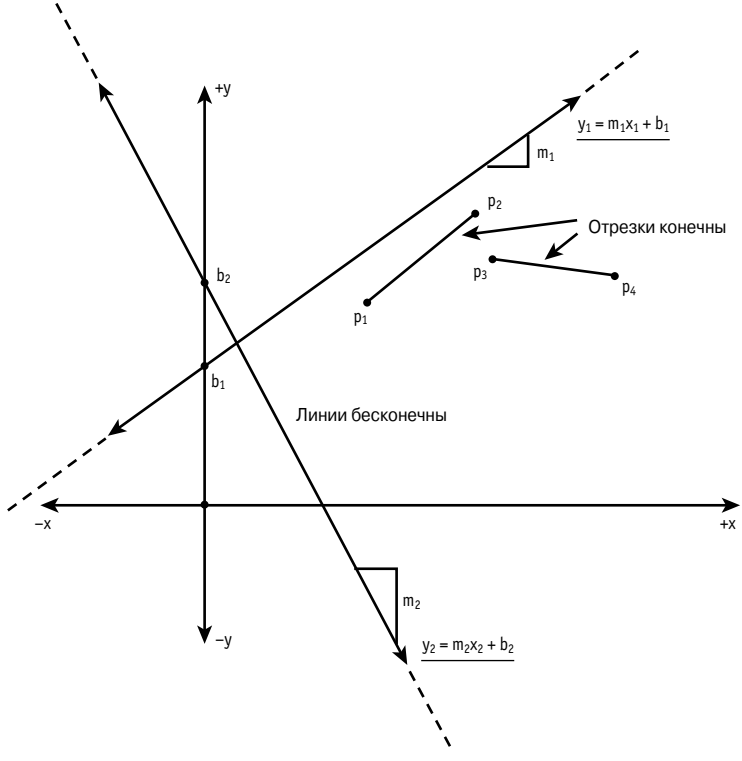


Рис. 13.27. Линии и отрезки

Рассмотрим следующую задачу. Объект движется с некоторой скоростью $\mathbf{V}_i$. Мы хотим определить, пересечет ли этот вектор некоторую плоскость (или линию). Поскольку объект движется со скоростью $\mathbf{V}_i$, через один кадр (или единицу времени) он будет находиться в положении $(x_0,y_0)+\mathbf{V}_i$ или, если рассматривать компоненты координат в отдельности:

$$x_1 = x_0 + v_{ix},$$
$$y_1 = y_0 + v_{iy}.$$

Таким образом, можно рассматривать вектор скорости как отрезок прямой линии, который указывает направление движения нашего объекта. Другими словами, мы хотим определить, существует ли точка пересечения (x,y) отрезков. Вот начальные условия:

$$\text{Отрезок объекта } \mathbf{S}_1 = \left\langle p_1(x_1, y_1) - p_0(x_0, y_0) \right\rangle$$
$$\text{Отрезок границы } \mathbf{S}_2 = \left\langle p_3(x_3, y_3) - p_2(x_2, y_2) \right\rangle$$

Нам необходимо точное значение точки пересечения с тем, чтобы при вычислении вектора отражения **F** знать его начальное положение (рис. 13.28). Задача кажется достаточно простой, но она гораздо сложнее, чем вы думаете. Пересечение двух прямых линий легко находится путем решения системы двух уравнений, но вопрос с пересечением отрезков решается не так просто. Это связано с тем, что, хотя отрезки и являются прямыми, они ограничены. Поэтому, даже если линии, определяемые отрезками, пересекаются, сами отрезки могут и не пересечься (рис. 13.29). Следовательно, необходимо не только определить точку пересечения линий, но и проверить, принадлежит ли эта точка обоим отрезкам.

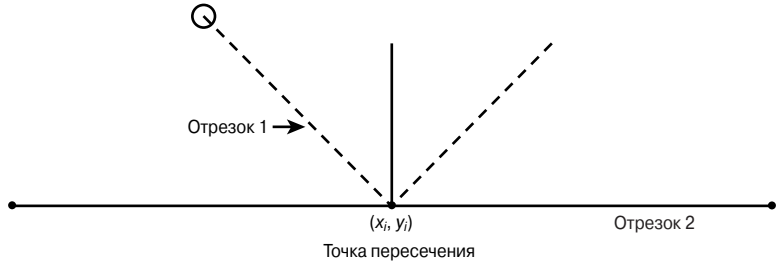


Рис. 13.28. Пересечение и отражение

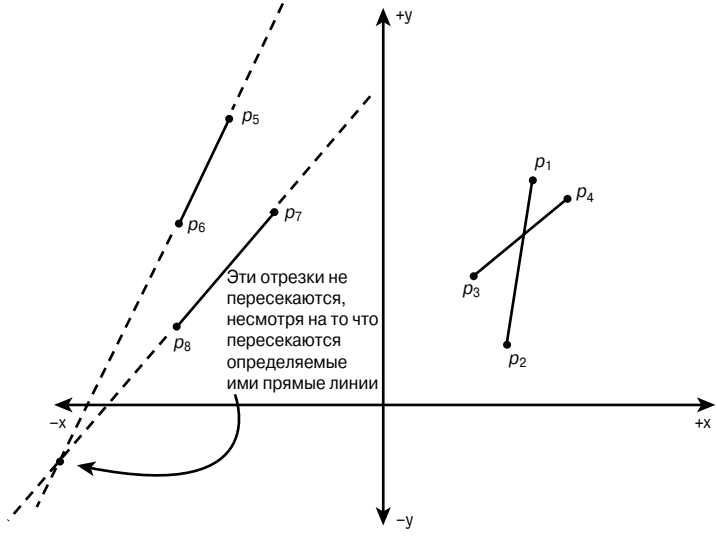


Рис. 13.29. Пересекающиеся и не пересекающиеся отрезки

Будем решать эту задачу с использованием параметрического представления каждого отрезка. Обозначим через **U** радиус-вектор точек отрезка $\mathbf{S}_1$, а **V** — радиус-вектор точек отрезка $\mathbf{S}_2$.

$$\mathbf{U} = p_0 + t \cdot \mathbf{S}_1, \quad 0 \le t \le 1$$
$$\mathbf{V} = p_2 + s \cdot \mathbf{S}_2, \quad 0 \le s \le 1$$

На рис. 13.30 показано, что именно представляют собой приведенные уравнения.

Теперь у нас есть все необходимое для окончательного решения поставленной задачи. Требуется найти значения параметров *s* и *t* и выяснить, находятся ли они в диапазоне от 0 до 1. Если да, то, подставляя значения данных параметров в уравнения, можно найти точку пересечения отрезков.

Итак, ищем решение (s, t) уравнения $\mathbf{U} = \mathbf{V}$, т.е. уравнения

$$s\mathbf{S}_2 - t\mathbf{S}_1 = p_0 - p_2 .$$

Расписывая это уравнение в проекциях на оси координат, получаем систему уравнений

$$\begin{cases} sS_{2x} - tS_{1x} = p_{0x} - p_{2x} \\ sS_{2y} - tS_{1y} = p_{0y} - p_{2y} \end{cases}$$

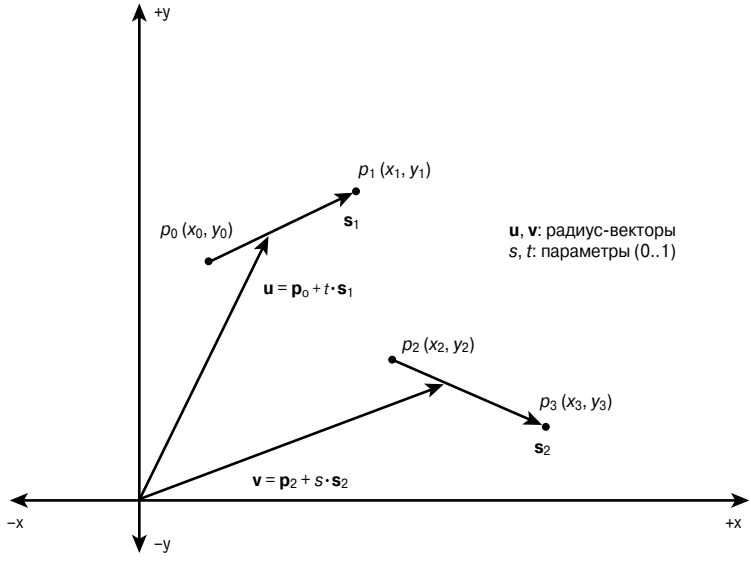


Рис. 13.30. Параметрическое представление отрезков

или в матричной форме:

$$\underbrace{\begin{bmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} s \\ t \end{bmatrix}}_{\mathbf{X}} = \underbrace{\begin{bmatrix} p_{0x} - p_{2x} \\ p_{0y} - p_{2y} \end{bmatrix}}_{\mathbf{B}}.$$

Используя для решения системы уравнений правило Крамера, получаем

$$s = \frac{\det\begin{bmatrix} p_{0x} - p_{2x} & -S_{1x} \\ p_{0y} - p_{2y} & -S_{1y} \end{bmatrix}}{\det\begin{bmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{bmatrix}}, \quad t = \frac{\det\begin{bmatrix} S_{2x} & p_{0x} - p_{2x} \\ S_{2y} & p_{0y} - p_{2y} \end{bmatrix}}{\det\begin{bmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{bmatrix}}.$$

Правило Крамера гласит, что вы можете решить систему уравнений $\mathbf{AX} = \mathbf{B}$, вычисляя $x_i = \det(\mathbf{A}_i)/\det(\mathbf{A})$. Здесь $\mathbf{A}_i$ — матрица, полученная заменой *i*-го столбца матрицы $\mathbf{A}$ вектором $\mathbf{B}$.

Определитель матрицы — достаточно сложно вычисляемая величина, но формулы для матриц размером 2×2 или 3×3 запомнить достаточно просто. Для матрицы 2×2 определитель вычисляется по следующей формуле:

$$\det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = (ad - bc).$$

После вычисления значений *s* и *t* необходимо проверить, попадают ли они в диапазон (0..1). Если хотя бы один из параметров не попадает в данный диапазон, исходные отрезки не пересекаются. Если же оба значения находятся в указанном диапазоне, отрезки пересекаются и точку пересечения можно получить, подставляя значения параметров в формулы для радиус-векторов.

Рассмотрим конкретный пример на рис. 13.31.

СЕКРЕТ

Если прямоугольники, окружающие отрезки, не перекрываются, то проверять пересечение этих отрезков нет необходимости — они не пересекаются.

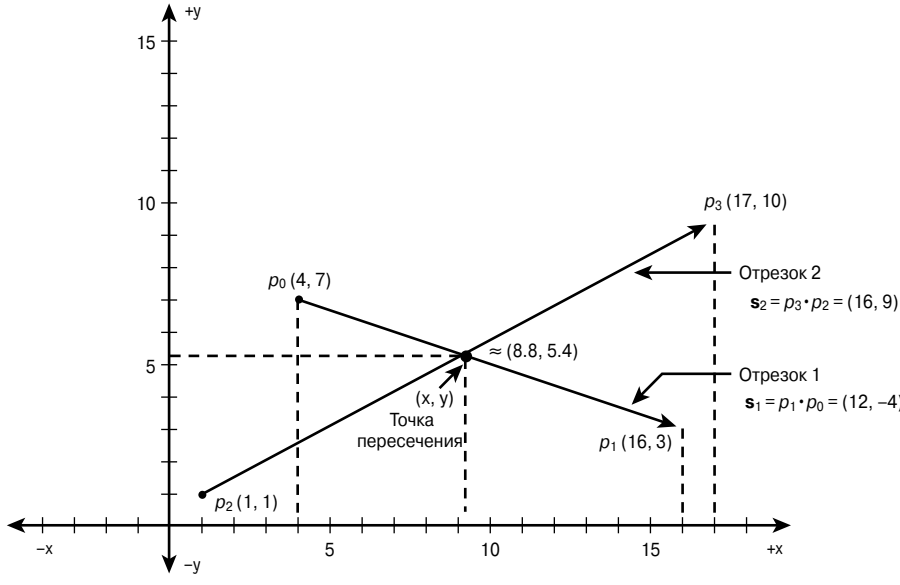


Рис. 13.31. Пример пересечения отрезков

Здесь

$$p_0 = (4,7), \quad p_1 = (16,3), \quad \mathbf{S}_1 = p_1 - p_0 = \langle 12,-4 \rangle,$$
$$p_2 = (1,1), \quad p_3 = (17,10), \quad \mathbf{S}_2 = p_3 - p_2 = \langle 16,9 \rangle.$$

Соответственно вычисляем значения параметров:

$$s = \frac{\det\begin{bmatrix} 3 & -12 \\ 6 & 4 \end{bmatrix}}{\det\begin{bmatrix} 16 & -12 \\ 9 & 4 \end{bmatrix}} = \frac{12+72}{64+108} \approx 0.488,$$

$$t = \frac{\det\begin{bmatrix} 16 & 3 \\ 9 & 6 \end{bmatrix}}{\det\begin{bmatrix} 16 & -12 \\ 9 & 4 \end{bmatrix}} = \frac{96-27}{64+108} \approx 0.401.$$

Подставляя параметры (которые попадают в диапазон (0..1), так что точка пересечения отрезков существует) в исходные уравнения для радиус-векторов, находим, что

$$\mathbf{U} = \langle 4,7 \rangle + 0.401 \cdot \langle 12,-4 \rangle \approx \langle 8.814, 5.395 \rangle,$$
$$\mathbf{V} = \langle 1,1 \rangle + 0.488 \cdot \langle 16,9 \rangle \approx \langle 8.814, 5.395 \rangle.$$

Как и следовало ожидать, радиус-векторы точки пересечения одинаковы.

Описанная технология использована в находящейся на прилагаемом компакт-диске демонстрационной программе `DEM013_7.CPP`, копия экрана которой показана на рис. 13.32. Попробуйте подредактировать код программы и изменить форму многоугольника.

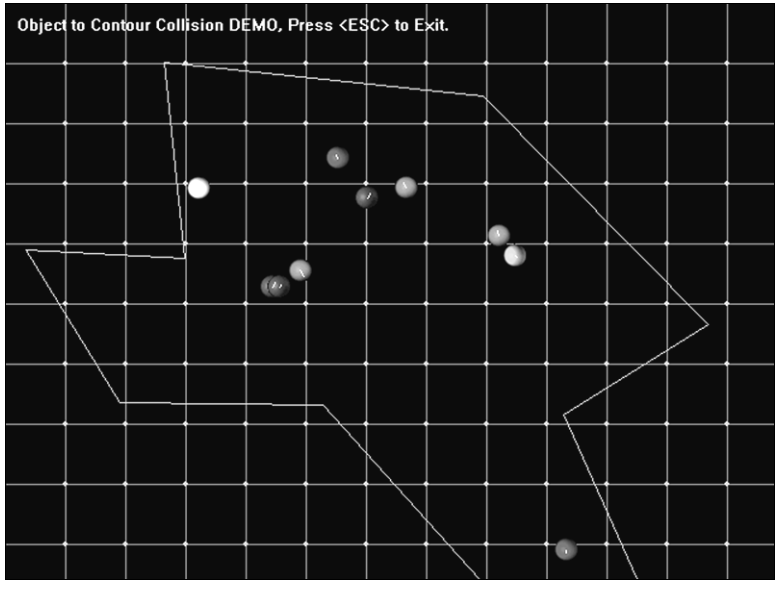


Рис. 13.32. Демонстрационная программа DEMO13_7.EXE

Реальное двухмерное столкновение объектов

Я специально отложил рассмотрение этого вопроса насколько мог, ибо при всей красоте и реалистичности картины столкновения объектов на экране компьютера за ней кроется огромный и отнюдь не простой труд. А теперь, когда я серьезно вас напугал, приступим.

На рис. 13.33 представлена задача, которую мы намерены решать. Имеется два объекта, моделируемых двухмерными окружностями (или трехмерными сферами), каждый из которых имеет свою массу и начальную скорость. Мы хотим вычислить их скорости после столкновения. Столкновение двух объектов уже рассматривалось в разделе "Законы сохранения", где, исходя из законов сохранения импульса

$$m_a v_{ai} + m_b v_{bi} = m_a v_{af} + m_b v_{bf}$$

и сохранения энергии

$$\frac{m_a v_{ai}^2}{2} + \frac{m_b v_{bi}^2}{2} = \frac{m_a v_{af}^2}{2} + \frac{m_b v_{bf}^2}{2},$$

мы получили скорости объектов после столкновения:

$$v_{af} = \frac{2m_b v_{bi} + (m_a - m_b) v_{ai}}{m_a + m_b},$$
$$v_{bf} = \frac{2m_a v_{ai} + (m_b - m_a) v_{bi}}{m_a + m_b}.$$

Эти уравнения совершенно справедливы для идеально упругого столкновения. Главная же их проблема в том, что они описывают одномерное столкновение. Что же необхо-

димо сделать, чтобы перейти к случаю двухмерного столкновения (например, шаров на бильярдном столе)?

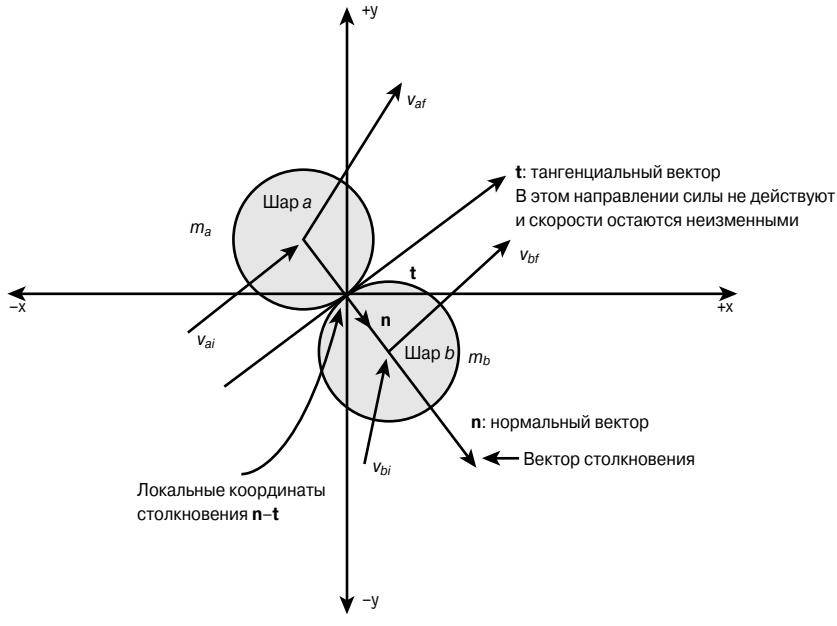


Рис. 13.33. Задача центрального столкновения двух тел

Рассмотрим упрощенную задачу столкновения одинаковых шаров, когда каждый шар имеет некоторую массу m; кроме того, мы полагаем, что шары симметричны и их центры масс располагаются в их центрах. (Мы также не рассматриваем возможное вращение шаров, считая, что вращательное движение отсутствует и шары летят в пространстве или скользят по плоскости. — *Прим. ред.*). Когда шары ударяются друг о друга, они ненадолго деформируются, часть кинетической энергии переходит в тепловую, нагревающую шары, часть — в потенциальную энергию деформации, которая после разделения шаров вновь переходит в кинетическую энергию их движения. Схематично процесс столкновения показан на рис. 13.34.

Столкновение состоит из двух фаз. Фаза деформации начинается в момент соприкосновения шаров, когда некоторое время шары движутся с одинаковой скоростью. По окончании фазы деформации начинается фаза восстановления, которая продолжается до тех пор, пока шары не разъединятся. Отсюда можно сделать вывод, что на самом деле столкновение шаров — весьма сложный физический процесс, который очень трудно смоделировать на компьютере, так что мы прибегнем к ряду упрощающих предположений. Даже при этих упрощениях наша модель оказывается очень близка к реальному столкновению и отлично выглядит на экране компьютера. Приведем эти предположения.

1. Время столкновения шаров очень мало (обозначим его как dt).
2. Во время столкновения положение шаров в пространстве не изменяется.
3. *Скорость шаров может претерпеть значительные изменения.*
4. В процессе столкновения никакие силы трения не действуют.

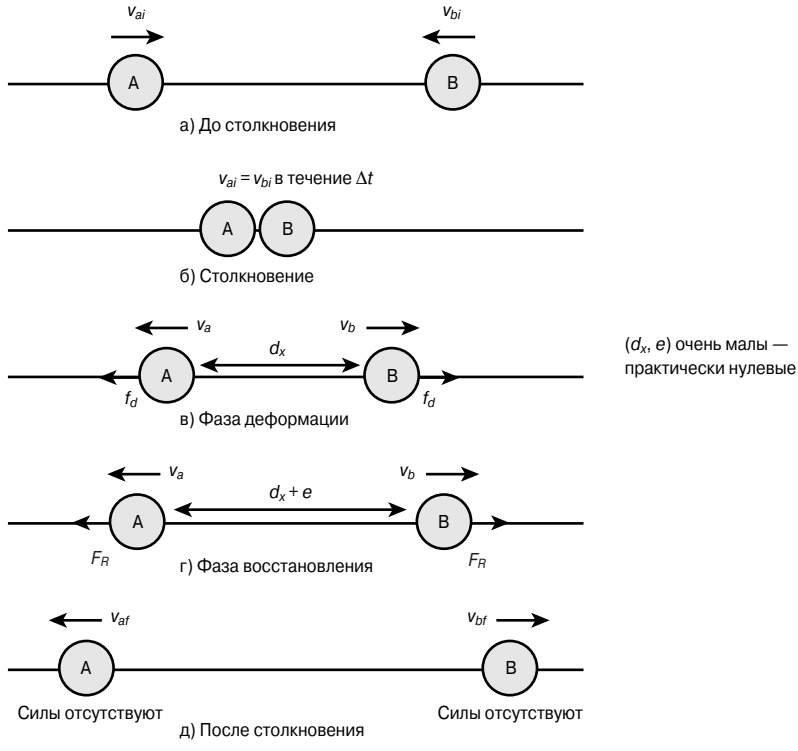


Рис. 13.34. Фазы столкновения

Я должен пояснить только третье предположение, поскольку все остальные кажутся предельно ясными. Чтобы оно было истинно, силы, действующие в момент соударения, должны быть мгновенными. Силы такого типа именуются импульсными, и именно в них находится ключ к решению задачи. Мы можем вычислить импульсы и, исходя из закона сохранения импульса, решить поставленную задачу. Математическое решение задачи достаточно сложное, так что я не буду приводить его полностью. Главное, что я хотел бы отметить, — это то, что в результате решения задачи появляется *коэффициент восстановления*, который описывает физическую модель столкновения:

$$e = \frac{v_{af} - v_{bf}}{v_{bi} - v_{ai}}. \qquad (1)$$

Этот коэффициент e, по сути, указывает идеальность столкновения. При $e = 1$ мы получаем модель абсолютно упругого столкновения, при $e = 0$ — абсолютно неупругого. Промежуточные значения e соответствуют реальным столкновениям шаров с потерей части кинетической энергии. Использование коэффициента e вместе с законом сохранения импульса

$$m_a v_{ai} + m_b v_{bi} = m_a v_{af} + m_b v_{bf}$$

дает нам следующие выражения для скоростей шаров после столкновения:

$$v_{af} = \frac{(1+e)m_b v_{bi} + (m_a - em_b)v_{ai}}{m_a + m_b},$$
$$v_{bf} = \frac{(1+e)m_a v_{ai} + (m_b - em_a)v_{bi}}{m_a + m_b}. \quad (2)$$

Вы не видите ничего замечательного? А ведь эти формулы очень похожи на формулы, полученные при использовании законов сохранения импульса и кинетической энергии. Попробуйте подставить значение $e = 1$, и вы получите те же формулы для скоростей после столкновения, что и ранее:

$$v_{af} = \frac{2m_b v_{bi} + (m_a - m_b)v_{ai}}{m_a + m_b},$$
$$v_{bf} = \frac{2m_a v_{ai} + (m_b - m_a)v_{bi}}{m_a + m_b}.$$

Так что пока мы вроде бы на верном пути. Главный минус рассмотренного решения состоит в том, что мы решили одномерную задачу, в то время как нас интересует двухмерное решение.

Возвращаясь к рис. 13.33, хочу обратить ваше внимание на оси координат, помеченные как **n** и **t**. Ось **n** (нормальная) идет в направлении линии столкновения, а ось **t** (тангенциальная) перпендикулярна ей. Считая, что мы знаем векторы, представляющие эти оси (о том, как их найти, я расскажу немного позже), можно записать ряд уравнений.

Первые уравнения, которые мы можем записать, связаны с тангенциальными составляющими скоростей до и после столкновения. Поскольку никаких сил трения нет и никакие импульсные силы вдоль тангенциальной оси не действуют (поверьте мне на слово), проекции импульсов (и скоростей) шаров вдоль тангенциальной оси остаются неизменными:

$$m_a \mathbf{v}_{ai}\mathbf{t} = m_a \mathbf{v}_{af}\mathbf{t},$$
$$m_b \mathbf{v}_{bi}\mathbf{t} = m_b \mathbf{v}_{bf}\mathbf{t}. \quad (3)$$

(Если хотите, можно скомбинировать эти два уравнения в одно:

$$m_a \mathbf{v}_{ai}\mathbf{t} + m_b \mathbf{v}_{bi}\mathbf{t} = m_a \mathbf{v}_{af}\mathbf{t} + m_b \mathbf{v}_{bf}\mathbf{t}\ .)$$

Понять мои записи очень легко: (a,b) относятся к шарам, (i,f) — к состояниям до столкновения и после него, а (n,t) — к компонентам вдоль нормальной и тангенциальной оси.

Поскольку массы шаров остаются после столкновения теми же, что и до него, следующие уравнения говорят о сохранении тангенциальной скорости шаров:

$$\mathbf{v}_{ai}\mathbf{t} = \mathbf{v}_{af}\mathbf{t},$$
$$\mathbf{v}_{bi}\mathbf{t} = \mathbf{v}_{bf}\mathbf{t}. \quad (4)$$

Половина задачи уже решена: мы знаем скорости шаров в тангенциальном направлении после столкновения. Осталось найти нормальные составляющие. Поскольку внешних сил нет, можно записать закон сохранения импульса вдоль нормальной оси:

$$m_a \mathbf{v}_{ai}\mathbf{n} + m_b \mathbf{v}_{bi}\mathbf{n} = m_a \mathbf{v}_{af}\mathbf{n} + m_b \mathbf{v}_{bf}\mathbf{n}\ . \quad (5)$$

И последнее соотношение, которое можно записать, — это значение коэффициента восстановления вдоль оси **n**:

$$e = \frac{\mathbf{v}_{af}\mathbf{n} - \mathbf{v}_{bf}\mathbf{n}}{\mathbf{v}_{bi}\mathbf{n} - \mathbf{v}_{ai}\mathbf{n}}. \tag{6}$$

Итак, что же мы имеем? У нас есть два уравнения, (5) и (6), и две неизвестные величины — нормальные составляющие скоростей шаров после удара. Мы можем найти эти составляющие... стоп! но они ведь уже найдены — формула (2) и есть решение приведенных уравнений! Итак, окончательно мы можем записать, что нормальные составляющие скоростей шаров после соударения равны

$$\begin{aligned} v_{af_n} &= \frac{(1+e)m_b\mathbf{v}_{bi}\mathbf{n} + (m_a - em_b)\mathbf{v}_{ai}\mathbf{n}}{m_a + m_b}, \\ v_{bf_n} &= \frac{(1+e)m_a\mathbf{v}_{ai}\mathbf{n} + (m_b - em_a)\mathbf{v}_{bi}\mathbf{n}}{m_a + m_b}. \end{aligned} \tag{7}$$

Система координат n-t

Теперь, когда у нас есть решение задачи столкновения, нам необходимо получить начальные значения нормальной и тангенциальной компонент векторов скорости, а затем, когда задача будет решена, преобразовать значения из системы координат **n-t** к системе координат x-y. Начнем с поиска векторов **n** и **t**.

Для того чтобы найти вектор **n**, достаточно сообразить, что это вектор единичной длины вдоль линии, соединяющей центры шаров A(x_{a0},y_{a0}) и B(x_{b0},y_{b0}). Найдем вектор между центрами шаров и нормализуем его:

$$\begin{aligned} \mathbf{N} &= B - A = \langle x_{b0} - x_{a0}, y_{b0} - y_{a0} \rangle, \\ \mathbf{n} &= \mathbf{N} / |\mathbf{N}| = \langle n_x, n_y \rangle. \end{aligned} \tag{8}$$

Теперь нужно найти тангенциальную ось **t**, перпендикулярную **n**. Для этого можно воспользоваться векторной геометрией, но лучше немного подумать, чтобы сообразить, что получить **t** можно вращением **n** на угол $\pi/2$ против часовой стрелки, т.е. $\mathbf{t} = \langle -n_y, n_x \rangle$ (рис. 13.35).

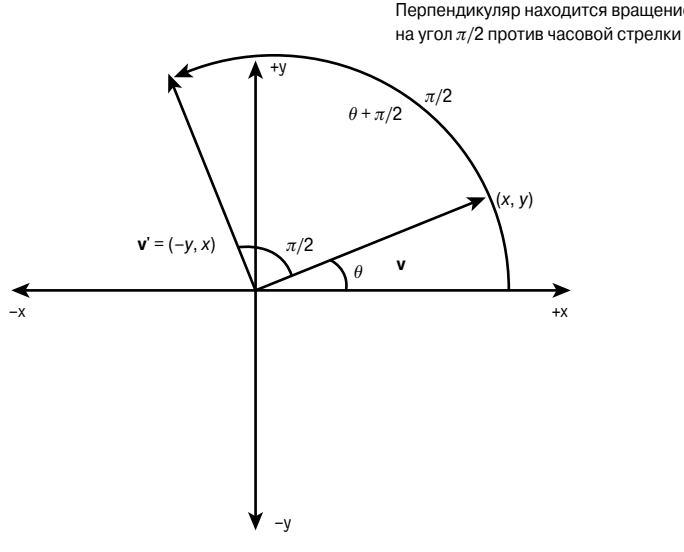


Рис. 13.35. Поиск перпендикулярного вектора

Теперь, когда у нас есть **n** и **t**, можно находить компоненты векторов скорости путем вычисления скалярных произведений, например вот как находится нормальная составляющая начальной скорости шара A:

$$\mathbf{v}_{ai}\mathbf{n} = \langle x_{vai}, y_{vai} \rangle \langle n_x, n_y \rangle = x_{vai} n_x + y_{vai} n_y .$$

Итак, вычислим все компоненты начальных скоростей шаров:

$$\begin{aligned}
\mathbf{v}_{ai}\mathbf{n} &= \langle x_{vai}, y_{vai} \rangle \langle n_x, n_y \rangle = x_{vai} n_x + y_{vai} n_y, \\
\mathbf{v}_{ai}\mathbf{t} &= \langle x_{vai}, y_{vai} \rangle \langle t_x, t_y \rangle = x_{vai} t_x + y_{vai} t_y, \\
\mathbf{v}_{bi}\mathbf{n} &= \langle x_{vbi}, y_{vbi} \rangle \langle n_x, n_y \rangle = x_{vbi} n_x + y_{vbi} n_y, \\
\mathbf{v}_{bi}\mathbf{t} &= \langle x_{vbi}, y_{vbi} \rangle \langle t_x, t_y \rangle = x_{vbi} t_x + y_{vbi} t_y .
\end{aligned} \tag{9}$$

Теперь у нас есть все для полного решения задачи. Опишем, как ее следует решать.

1. Вычислить **n** и **t** в соответствии с формулой (8).
2. Найти все нормальные и тангенциальные компоненты v_{ai} и v_{bi}, используя формулу (9).
3. Подставить полученные значения в уравнения (7) и получить нормальные составляющие скоростей после столкновения (тангенциальные составляющие остаются теми же, что и до него).
4. Преобразовать полученные результаты из системы координат **n-t** в систему координат x-y.

Шаг 4 остается в качестве домашнего упражнения, а пока поговорим о тензорах. Шучу, шучу! Не брошу же я вас в самом конце пути, правда?

Итак, вот как можно записать скорости шаров A и B после столкновения в координатах **n-t**:

$$\begin{aligned}
\mathbf{v}_{af} &= \langle \mathbf{v}_{af}\mathbf{n}, \mathbf{v}_{af}\mathbf{t} \rangle, \\
\mathbf{v}_{bf} &= \langle \mathbf{v}_{bf}\mathbf{n}, \mathbf{v}_{bf}\mathbf{t} \rangle .
\end{aligned}$$

Теперь забудем о столкновениях и подумаем о векторной геометрии. Взгляните на рис. 13.36 — на нем схематично изображена наша задача.

Итак, у нас есть вектор в одной системе координат и мы хотим записать его компоненты в другой системе координат. Как? Для этого опять воспользуемся скалярным произведением. Взгляните на вектор **v** на рис. 13.36. Этот вектор можно представить как сумму проекций на оси **n** и **t**; для получения проекций вектора на оси x и y необходимо суммировать соответствующие проекции **n**- и **t**-компонентов исходного вектора. Все проекции можно найти при помощи скалярного произведения векторов. Так, для шара A находим:

$$\begin{aligned}
\mathbf{v}_{af} &= (\mathbf{v}_{af}\mathbf{n})\mathbf{n} + (\mathbf{v}_{af}\mathbf{t}) \\
&= (\mathbf{v}_{af}\mathbf{n}) \langle n_x, n_y \rangle + (\mathbf{v}_{af}\mathbf{t}) \langle t_x, t_y \rangle,
\end{aligned}$$

откуда

$$\begin{aligned}
x_{af} &= \mathbf{v}_{af} \langle 1, 0 \rangle = (\mathbf{v}_{af}\mathbf{n}) n_x + (\mathbf{v}_{ax}\mathbf{t}) t_x, \\
y_{af} &= \mathbf{v}_{af} \langle 0, 1 \rangle = (\mathbf{v}_{af}\mathbf{n}) n_y + (\mathbf{v}_{ax}\mathbf{t}) t_y .
\end{aligned}$$

Аналогично для шара B получаем:

$$\begin{aligned}\mathbf{v}_{bf} &= \left(\mathbf{v}_{bf}\mathbf{n}\right)\mathbf{n} + \left(\mathbf{v}_{bf}\mathbf{t}\right) = \\ &= \left(\mathbf{v}_{bf}\mathbf{n}\right)\left\langle n_x, n_y\right\rangle + \left(\mathbf{v}_{bf}\mathbf{t}\right)\left\langle t_x, t_y\right\rangle\end{aligned}$$

и

$$\begin{aligned}x_{bf} &= \mathbf{v}_{bf}\left\langle 1,0\right\rangle = \left(\mathbf{v}_{bf}\mathbf{n}\right)n_x + \left(\mathbf{v}_{bx}\mathbf{t}\right)t_x, \\ y_{bf} &= \mathbf{v}_{bf}\left\langle 0,1\right\rangle = \left(\mathbf{v}_{bf}\mathbf{n}\right)n_y + \left(\mathbf{v}_{bx}\mathbf{t}\right)t_y.\end{aligned}$$

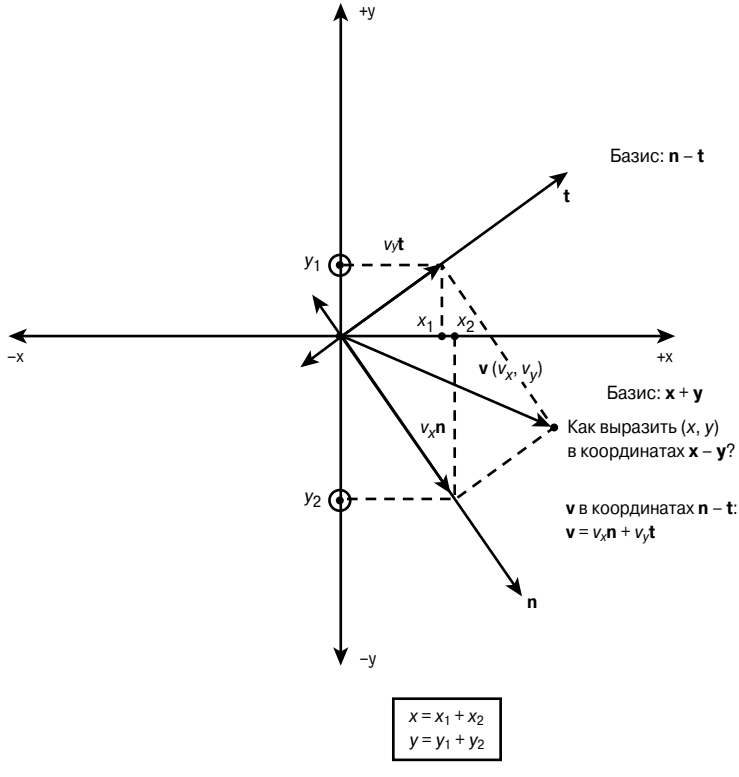


Рис. 13.36. Преобразование вектора от одного базиса к другому

Теперь вам осталось назначить шарам в программе новые скорости — и дело сделано. Так как мне кажется, что разобраться с кодом проще, чем с математикой, я решил полностью привести исходный код функции из демонстрационной программы `DEMO13_8.CPP`, находящейся на прилагаемом компакт-диске. В этой программе (копия ее экрана показана на рис. 13.37) моделируются столкновения ряда шаров. В нижней части экрана выводится значение суммарной кинетической энергии системы. Вы можете изменять значение коэффициента восстановления и следить при этом за изменением суммарной кинетической энергии, которая убывает при $e < 1$, остается неизменной при $e = 1$ и растет при $e > 1$ (и почему нельзя сделать то же с моим банковским счетом?).

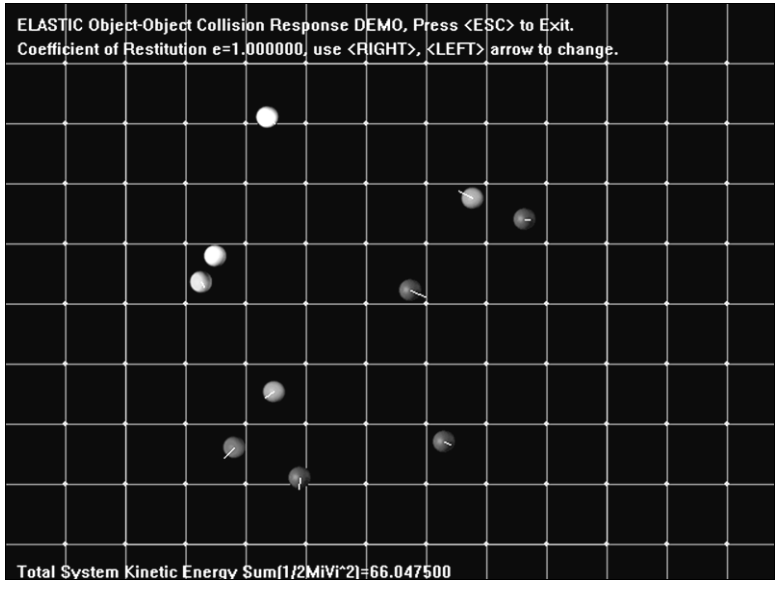


Рис. 13.37. Демонстрационная программа DEMO13_8.EXE

```
void Collision_Response(void)
{
// Эта функция выполняет моделирование
// столкновений между шарами

// Мы знаем, что
// va2 = (e+1)*mb*vb1+va1(ma - e*mb)/(ma+mb)
// vb2 = (e+1)*ma*va1+vb1(ma - e*mb)/(ma+mb)

// Кроме того, мы знаем, что вектор нормали при
// соударении вычисляется исходя из сферичности
// объектов, т.е. соединяет их центры

// Шаг 1. Попарно проверяем наличие столкновений шаров.
// Имеются и лучшие алгоритмы, чем двойной вложенный цикл,
// но объектов немного, так что такое простое решение
// вполне приемлемо
for (int ball_a = 0; ball_a < NUM_BALLS; ball_a++)
   {
   for (int ball_b = ball_a+1; ball_b < NUM_BALLS; ball_b++)
      {
      if (ball_a == ball_b) continue;

      // Вычисляем нормальный вектор a->b
      float nabx = (balls[ball_b].varsF[INDEX_X] -
                    balls[ball_a].varsF[INDEX_X] );
      float naby = (balls[ball_b].varsF[INDEX_Y] -
                    balls[ball_a].varsF[INDEX_Y] );
      float length = sqrt(nabx*nabx + naby*naby);
```

```
// Столкновение?
if (length <= 2.0*(BALL_RADIUS*0.75))
   {
   // Шары столкнулись

   // Нормализуем вектор нормали и
   // вычисляем тангенциальный вектор
   nabx/=length;
   naby/=length;
   float tabx =  -naby;
   float taby =  nabx;

   // Изображаем столкновение
   DDraw_Lock_Primary_Surface();

   // Синий цвет— для нормальной оси
   Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
     balls[ball_a].varsF[INDEX_Y]+0.5,
     balls[ball_a].varsF[INDEX_X]+20*nabx+0.5,
     balls[ball_a].varsF[INDEX_Y]+20*naby+0.5,
     252, primary_buffer, primary_lpitch);

   // Желтый - для тангенциальной оси
   Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
     balls[ball_a].varsF[INDEX_Y]+0.5,
     balls[ball_a].varsF[INDEX_X]+20*tabx+0.5,
     balls[ball_a].varsF[INDEX_Y]+20*taby+0.5,
     251, primary_buffer, primary_lpitch);

   DDraw_Unlock_Primary_Surface();


   // Шаг 2. Вычисление начальных скоростей

   float vait = DOT_PRODUCT(
      balls[ball_a].varsF[INDEX_XV],
      balls[ball_a].varsF[INDEX_YV],
      tabx, taby);

   float vain = DOT_PRODUCT(
      balls[ball_a].varsF[INDEX_XV],
      balls[ball_a].varsF[INDEX_YV],
      nabx, naby);

   float vbit = DOT_PRODUCT(
      balls[ball_b].varsF[INDEX_XV],
      balls[ball_b].varsF[INDEX_YV],
      tabx, taby);

   float vbin = DOT_PRODUCT(
      balls[ball_b].varsF[INDEX_XV],
      balls[ball_b].varsF[INDEX_YV],
      nabx, naby);
```

```
        // Шаг 3. Вычисляем скорости шаров после
        // столкновения. Примечание: этот код легко
        // оптимизировать, я просто хочу расписать
        // все подробно

        float ma = balls[ball_a].varsF[INDEX_MASS];
        float mb = balls[ball_b].varsF[INDEX_MASS];

        float vafn = (mb*vbin*(cof_E+1) +
            vain*(ma - cof_E*mb)) / (ma + mb);
        float vbfn = (ma*vain*(cof_E+1) -
            vbin*(ma - cof_E*mb)) / (ma + mb);

        // Тангенциальные компоненты не изменяются
        float vaft = vait;
        float vbft = vbit;

        // Вся проблема в том, что мы не в той системе
        // координат и нам надо преобразовать
        // полученные результаты к координатам xy

        float xfa = vafn*nabx + vaft*tabx;
        float yfa = vafn*naby + vaft*taby;

        float xfb = vbfn*nabx + vbft*tabx;
        float yfb = vbfn*naby + vbft*taby;

        // Сохранить результаты
        balls[ball_a].varsF[INDEX_XV] = xfa;
        balls[ball_a].varsF[INDEX_YV] = yfa;

        balls[ball_b].varsF[INDEX_XV] = xfb;
        balls[ball_b].varsF[INDEX_YV] = yfb;

        // Обновить положения
        balls[ball_a].varsF[INDEX_X]+=
           balls[ball_a].varsF[INDEX_XV];
        balls[ball_a].varsF[INDEX_Y]+=
           balls[ball_a].varsF[INDEX_YV];

        balls[ball_b].varsF[INDEX_X]+=
           balls[ball_b].varsF[INDEX_XV];
        balls[ball_b].varsF[INDEX_Y]+=
           balls[ball_b].varsF[INDEX_YV];

        } // if

      } // for ball2

    } // for ball1

} // Collision_Response
```

Простая кинематика

Термин *кинематика* означает множество вещей. Художник понимает под этим словом одно, программист другое, и оба понимают под ним совсем не то, что физик. В этом разделе книги под кинематикой мы подразумеваем механику движения связанных звеньев жестких конструкций. В компьютерной анимации есть две кинематические задачи — *прямая* и *обратная*. Прямая задача показана на рис. 13.38. Здесь вы видите последовательно связанные звенья жесткой конструкции (обычные рычаги). Каждый рычаг может свободно вращаться в плоскости, тем самым обеспечивая две степени свободы конструкции, показанной па рисунке: θ_1 и θ_2. Кроме того, известны длины рычагов l_1 и l_2. Прямая задача кинематики состоит в определении положения p_2 по заданным значениям θ_1, θ_2, l_1 и l_2.

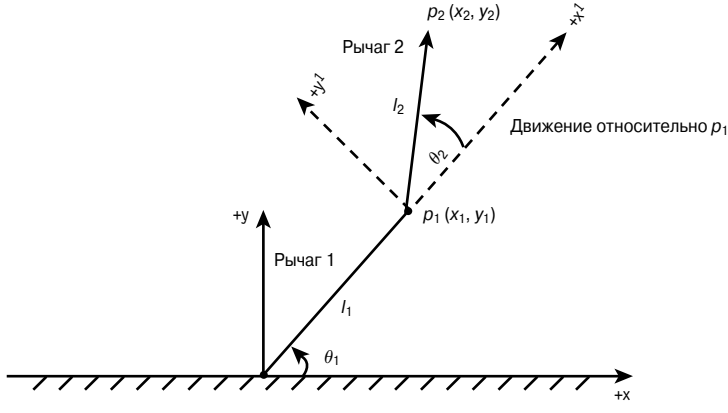


Рис. 13.38. Прямая задача кинематики

Почему может потребоваться решать такую задачу? Если вы пишете игру, в которой хотите добиться реального движения составных объектов, вряд ли вы стали бы задавать такой вопрос. Например, трехмерную анимацию можно осуществить двумя путями. Быстрый и грубый метод состоит в том, чтобы иметь набор сеток, представляющих трехмерную анимацию объекта. Более гибкий состоит в наличии одной сетки, в которой имеется ряд соединений и рычагов, и в реальном времени моделировать движение этого объекта. Но для такого моделирования вы должны понимать, как именно его осуществить и как, например, будут двигаться кисти рук относительно локтей, локти относительно плеч и т.д.

Вторая задача кинематики обратна первой. По данным координатам точки p_2 нужно найти значения θ_1 и θ_2, которые удовлетворяют условиям l_1 и l_2 физической модели. Эта задача гораздо сложнее, чем вы можете себе представить (на рис. 13.39 показано, с чем это связано).

На рис. 13.39 вы видите два разных решения, удовлетворяющих поставленной задаче (не считая того, что при некоторых данных решения вообще может не существовать, в то время как решение прямой задачи существует всегда. — *Прим. ред.*). Я не буду касаться этой проблемы в общем случае, поскольку в большинстве случаев нам не придется решать данную задачу, но позже я покажу, как ее можно обойти.

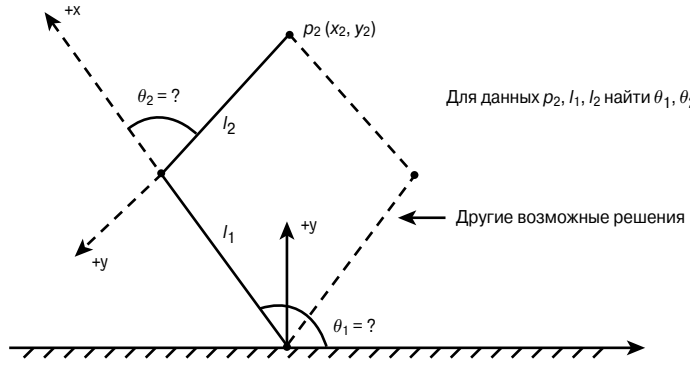


Рис. 13.39. Обратная задача кинематики

Решение прямой задачи кинематики

Сейчас я хочу показать вам, как просто решается прямая задача кинематики. Вернитесь к рис. 13.38. Как видите, перед нами простая задача относительного движения. Если посмотреть на ситуацию с точки зрения соединения 2, то положение точки p_2 определяется ее перемещением на расстояние l_2 и вращением на угол θ_2. Однако положение самой точки p_1 определяется ее перемещением на расстояние l_1 и вращением на угол θ_1. Следовательно, получить решение задачи можно путем последовательных перемещений и вращений от связи к связи. Будем решать задачу по частям.

Забудем о первом рычаге и будем рассматривать второй. В данном случае начальной точкой является точка p_1, которую мы хотим переместить на расстояние l_2 вдоль оси X и повернуть в плоскости XY на угол θ_2. Это совсем несложно — нужно лишь применить следующие преобразования к p_1: $p_2 = p_1 * \mathbf{T}_{l2} * \mathbf{R}_{\theta 2}$. Что? Нам неизвестна точка p_1? Неважно — пока мы просто считаем, что знаем ее координаты. $\mathbf{T}_{l2}$ и $\mathbf{R}_{\theta_2}$ представляют собой стандартные матрицы двухмерного перемещения и вращения, с которыми мы встречались в главе 8, "Растеризация векторов и двухмерные преобразования". Итак, мы имеем

$$\mathbf{T}_{l2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_2 & 0 & 1 \end{bmatrix}, \ \mathbf{R}_{\theta 2} = \begin{bmatrix} \cos\theta_2 & \sin\theta_2 & 0 \\ -\sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

и соответственно p_2 вычисляется как произведение

$$p_2 = p_1 * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_2 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta_2 & \sin\theta_2 & 0 \\ -\sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Аналогично поиску p_2 можно найти и p_1, применив соответствующие преобразования к p_0 — перенося ее на l_1 и поворачивая на θ_1:

$$p_2 = p_0 * \mathbf{T}_{l2} * \mathbf{R}_{\theta 2} * \mathbf{T}_{l1} * \mathbf{R}_{\theta 1}, \text{ где } p_0 = [0,0,1].$$

Причина наличия третьей компоненты в двухмерном случае в том, что таким образом мы в состоянии обеспечить гомогенное преобразование и выполнить перемещение при помощи матричного умножения. Поэтому все точки, с которыми мы имеем дело в нашем случае, имеют вид (x, y, 1). Матрицы $\mathbf{T}_{l1}$ и $\mathbf{R}_{\theta_1}$ имеют тот же вид, что и матрицы $\mathbf{T}_{l2}$ и $\mathbf{R}_{\theta_2}$, но с другими значениями. Обратите внимание на порядок умножения: поскольку мы рабо-

таем в обратном направлении, то сначала должны преобразовать p_0 с помощью $\mathbf{T}_{l2} * \mathbf{R}_{\theta 2}$, а затем — $\mathbf{T}_{l1} * \mathbf{R}_{\theta 1}$. Порядок умножения играет роль!

Нетрудно понять, что в общем случае множества соединений общее решение прямой задачи кинематики для n звеньев будет иметь вид

$$p_n = p_o * \mathbf{T}_{\text{n}} * \mathbf{R}_{\text{n}} * \mathbf{T}_{\text{n-1}} * \mathbf{R}_{\text{n-1}} * \mathbf{T}_{\text{n-2}} * \mathbf{R}_{\text{n-2}} * \ldots * \mathbf{T}_1 * \mathbf{R}_1.$$

Чтобы убедиться, что этот метод корректно работает, применим его к конкретной задаче, приведенной на рис. 13.40.

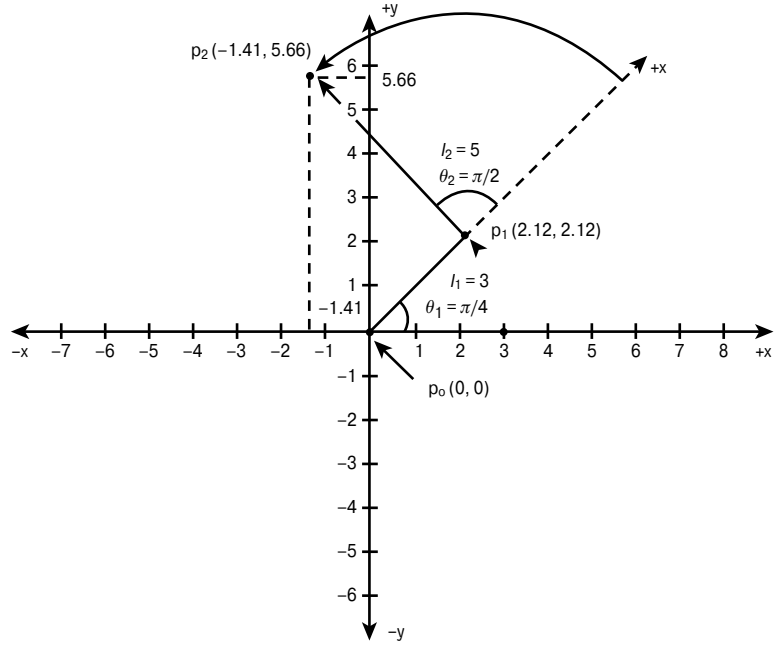


Рис. 13.40. Решение задачи кинематики "на бумаге"

Итак, в приведенной задаче

$$l_1 = 3, l_2 = 5, \theta_1 = \pi/4, \theta_2 = \pi/2,$$
$$p_0 = (0,0).$$

Теперь посмотрим, какой ответ дают точные математические вычисления:

$$p_2 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0.7071 & 0.7071 & 0 \\ -0.7071 & 0.7071 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \right)$$
$$= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} * \left(\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 5 & 1 \end{bmatrix} * \begin{bmatrix} 0.7071 & 0.7071 & 0 \\ -0.7071 & 0.7071 & 0 \\ 2.1213 & 2.1213 & 1 \end{bmatrix} \right)$$
$$= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -0.7071 & 0.7071 & 0 \\ -0.7071 & -0.7071 & 0 \\ -1.4142 & 5.6568 & 1 \end{bmatrix} = \begin{bmatrix} -1.4142 & 5.6568 & 1 \end{bmatrix}.$$

Итак, вычисленный ответ совпадает с показанным на рисунке и равен p_2(-1.4142, 5.6568). Как видите, решить двухмерную задачу не так сложно. Конечно, ре-

шение трехмерной задачи существенно осложняется наличием оси Z, тем не менее принципы решения остаются теми же.

На прилагаемом компакт-диске находится демонстрационная программа DEMO13_9.CPP, в которой использована описанная модель решения прямой задачи кинематики (копия экрана программы показана на рис. 13.41).

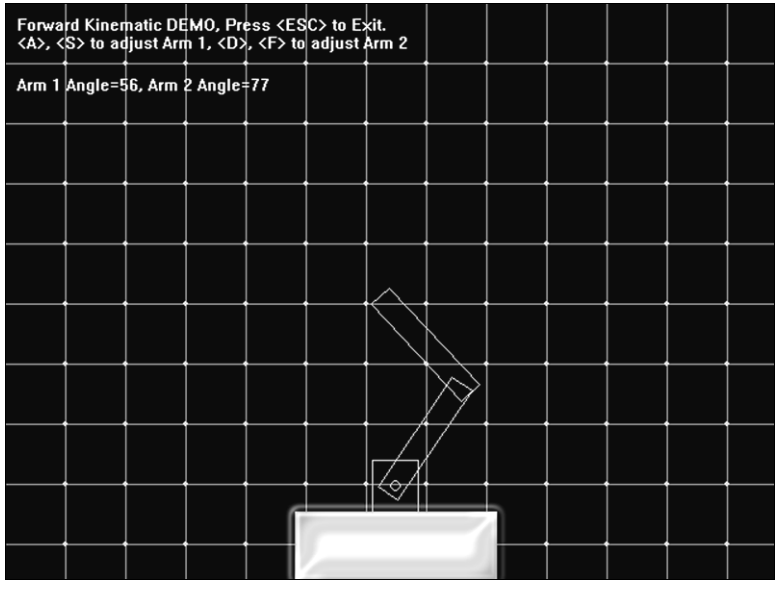


Рис. 13.41. Демонстрационная программа DEMO13_9.EXE

Решение обратной задачи кинематики

Решение обратной задачи в общем случае гораздо сложнее. Я не буду решать ее, я только хочу наметить дорогу, по которой следует идти, если вы решитесь на это. В предыдущем разделе была решена прямая задача кинематики: по заданным p_0, l_1, l_2, θ_1 и θ_2 найдено p_2. Но что делать, если не известно ни θ_1, ни θ_2, но известно p_2? Решение можно найти путем записи уравнений преобразований и поиска неизвестных углов. Проблема в том, что получающаяся система уравнений может иметь несколько решений. Следовательно, для поиска конкретного решения необходимо ввести дополнительные ограничения или использовать эвристические методы.

В качестве примера рассмотрим простейшую задачу с одной связью, показанную на рис. 13.42. Здесь имеется одна связь l_1 под углом θ_1 к оси X. Как определить θ_1, если известно $p_1(x_1,y_1)$?

Можно воспользоваться матричным уравнением для прямой задачи кинематики:

$$
\begin{aligned}
p_1 &= p_0 * \mathbf{T}_{l1} * \mathbf{R}_{\theta 1} = \\
&= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \\
&= \begin{bmatrix} l_1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.
\end{aligned}
$$

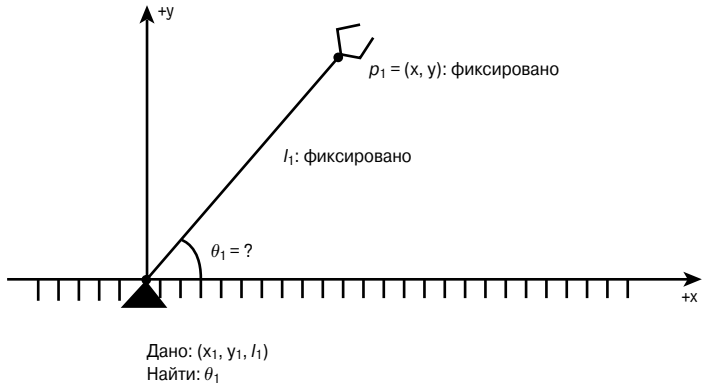


Рис. 13.42. Пример обратной задачи кинематики с одной связью

Таким образом, $p_1(x_1, y_1) = [l_1 \cos\theta_1 \quad l_1 \sin\theta_1 \quad 1]$, т.е.

$$x_1 = l_1 \cos\theta_1,$$
$$y_1 = l_1 \sin\theta_1,$$

соответственно $\theta_1 = \arccos(x_1/l_1)$ (или $\theta_1 = \arcsin(y_1/l_1)$).

Я мог бы решать задачу в матричном виде до конца, но обычная запись более иллюстративна.

Полученная система уравнений избыточна. Другими словами, для определения искомого угла достаточно одного уравнения; более того, если известна одна координата p_1, то вторую координату можно вычислить из второго уравнения. Достаточно немного подумать, чтобы понять, с чем это связано: рычаг l_1 приводит к потере точкой p_1 одной степени свободы, так что эта точка не может располагаться где угодно. При наличии же двух или большего числа связей очевидно, что задача может иметь несколько решений.

Системы частиц

Это очень сложная тема. Система частиц может быть и крайне сложной, и очень простой. В основном системы частиц представляют собой физические модели множества мелких частиц. Такие модели хороши для имитации взрывов, реактивных следов и т.п. Вы уже знаете достаточно, чтобы суметь создать собственную систему частиц, я только хочу немного помочь вам, показав, как создать очень быструю и простую систему частиц с размером в один пиксель.

Допустим, необходимо создать частицы для имитации взрыва. Поскольку система частиц — это не что иное, как n частиц, давайте сконцентрируемся на модели одной частицы.

Что требуется для каждой частицы

Если хотите, вы можете моделировать столкновения частиц, передачу импульса и прочие физические явления, но в большинстве своем системы частиц используют очень простые модели. Вот основные возможности модели системы частиц:

- положение;
- скорость;

- цвет/анимация;
- время жизни;
- гравитация;
- ветер.

Когда вы начинаете работу с частицей, то задаете ее положение в пространстве, начальную скорость, цвет и время жизни. Частица может представлять светящийся осколок или искру, в таком случае потребуется анимация ее цвета. Кроме того, вы можете предусмотреть учет глобальных сил, действующих на все частицы системы, таких, как гравитация или ветер. Вероятно, вы захотите иметь специальную функцию, которая создает коллекции частиц с заданными начальными свойствами. И, разумеется, для большего реализма вы захотите, чтобы частицы отскакивали от объектов в игре, причем делали это более-менее реалистично (хотя большую часть времени жизни частицы будут просто перемещаться в пространстве, не взаимодействуя с остальными объектами).

Разработка процессора частиц

Для создания системы частиц требуются три отдельные вещи.

- Структура данных для описания частицы.
- Процессор, который обрабатывает движение каждой частицы.
- Функция для генерации начальных данных каждой частицы.

Начнем со структуры данных. Я предполагаю, что используется 8-битовый цвет, так что для представления цвета мне нужен один байт, а не RGB-цвет. Кстати, эффекты, связанные с цветами, легче реализовать с использованием 8-битового цвета. Конечно, особой трудности в использовании, например, 16-битовых цветов нет, но я использую именно 8-битовые, чтобы сосредоточиться на работе с частицами, а не их цветом. Итак, вот первая попытка создания структуры данных для одной частицы:

```
// Отдельная частица
typedef struct PARTICLE_TYP
{
  int   state;          // Состояние частицы
  int   type;           // Тип частицы
  float x,y;            // Положение частицы
  float xv,yv;          // Скорость частицы
  int   curr_color;     // Текущий цвет частицы
  int   start_color;    // Начальный цвет
  int   end_color;      // Конечный цвет
  int   counter;        // Таймер времени жизни
  int   max_count;      // Максимальное значение таймера
} PARTICLE, *PARTICLE_PTR;
```

Добавим некоторые глобальные переменные для работы с внешними силами, такими, как гравитация и ветер.

```
float particle_wind    = 0;       // В направлении X
float particle_gravity = 0.02;    // В направлении Y
```

Определим также ряд символьных констант для реализации некоторых эффектов.

```
// Состояние частицы
#define PARTICLE_STATE_DEAD         0
#define PARTICLE_STATE_ALIVE        1
```

```
// Типы частиц
#define PARTICLE_TYPE_FLICKER    0
#define PARTICLE_TYPE_FADE       1

// Цвета частиц
#define PARTICLE_COLOR_RED       0
#define PARTICLE_COLOR_GREEN     1
#define PARTICLE_COLOR_BLUE      2
#define PARTICLE_COLOR_WHITE     3

#define MAX_PARTICLES          256

// Диапазоны цветов
#define COLOR_RED_START         32
#define COLOR_RED_END           47

#define COLOR_GREEN_START       96
#define COLOR_GREEN_END        111

#define COLOR_BLUE_START       144
#define COLOR_BLUE_END         159

#define COLOR_WHITE_START       16
#define COLOR_WHITE_END         31
```

Надеюсь, вы поняли мою задумку? Я хочу получить частицы разных цветов — красные, синие, зеленые и белые, которые к тому же подразделяются на два типа: гаснущие и вспыхивающие. Внести изменения при работе с 16-битовым цветом в данные определения очень легко, так что при необходимости сделайте это самостоятельно.

И наконец: одна маленькая частица — это хорошо, но много частиц — лучше, поэтому необходимо обеспечить место для их хранения.

```
PARTICLE particles[MAX_PARTICLES];
```

Теперь можно приступать к разработке функции, работающей с каждой частицей системы.

Программный процессор частиц

Нам необходимы функция для инициализации всех частиц, их настройки, обработки и освобождения всех ресурсов по завершении работы. Начнем с функции инициализации.

```
void Init_Reset_Particles(void)
{
// Эта функция служит как для начальной инициализации,
// так и для сброса системы частиц

// Циклически сбрасываем состояние всех частиц
for (int index=0; index<MAX_PARTICLES; index++)
{
  particles[index].state = PARTICLE_STATE_DEAD;
  particles[index].type  = PARTICLE_TYPE_FADE;
  particles[index].x     = 0;
  particles[index].y     = 0;
  particles[index].xv    = 0;
```

```
  particles[index].yv    = 0;
  particles[index].start_color = 0;
  particles[index].end_color   = 0;
  particles[index].curr_color  = 0;
  particles[index].counter     = 0;
  particles[index].max_count   = 0;
  } // if

} // Init_Reset_Particles
```

Эта функция присваивает всем полям всех частиц нулевые значения и подготавливает их к использованию. Если в вашей конкретной системе требуется какая-либо дополнительная инициализация, разместите ее код в этой функции. Следующая функция, которая нам необходима, — это функция, которая устанавливает начальные значения параметров частиц. В настоящий момент я представляю вам функцию, которая просто находит свободную частицу и, если таковая есть, настраивает соответствующие поля структуры данных переданными в качестве параметров значениями.

```
void Start_Particle(int type, int color, int count,
                int x, int y, int xv, int yv)
{
int pindex = -1; // Индекс частицы

// Находим свободную частицу
for (int index=0; index < MAX_PARTICLES; index++)
  if (particles[index].state == PARTICLE_STATE_DEAD)
    {
    // Устанавливаем ее индекс
    pindex = index;
    break;
    } // if

// Свободных частиц нет
if (pindex==-1)
  return;

// Устанавливаем основные характеристики
particles[pindex].state = PARTICLE_STATE_ALIVE;
particles[pindex].type  = type;
particles[pindex].x     = x;
particles[pindex].y     = y;
particles[pindex].xv    = xv;
particles[pindex].yv    = yv;
particles[pindex].counter    = 0;
particles[pindex].max_count  = count;

// Определяем диапазоны цвета
switch(color)
{
 case PARTICLE_COLOR_RED:
  {
  particles[pindex].start_color = COLOR_RED_START;
  particles[pindex].end_color   = COLOR_RED_END;
```

```
   } break;
  case PARTICLE_COLOR_GREEN:
   {
   particles[pindex].start_color = COLOR_GREEN_START;
   particles[pindex].end_color   = COLOR_GREEN_END;
   } break;
  case PARTICLE_COLOR_BLUE:
   {
   particles[pindex].start_color = COLOR_BLUE_START;
   particles[pindex].end_color   = COLOR_BLUE_END;
   } break;
  case PARTICLE_COLOR_WHITE:
   {
   particles[pindex].start_color = COLOR_WHITE_START;
   particles[pindex].end_color   = COLOR_WHITE_END;
   } break;
  break;
} // switch

// Тип частицы
if (type == PARTICLE_TYPE_FLICKER)
{
   // Устанавливаем текущий цвет
   particles[index].curr_color  =
      RAND_RANGE(particles[index].start_color,
            particles[index].end_color);

} // if
else
{
   // Гаснущая частица — назначаем текущий цвет
   particles[index].curr_color =
      particles[index].start_color;
} // if

} // Start_Particle
```

НА ЗАМЕТКУ В этой функции нет проверки на корректность работы. Я просто не озаботился ею — в конце концов, если вы не сможете создать частицу-другую, особого вреда от этого не будет. Однако вы можете создать более интеллектуальную и надежную функцию.

Для создания частицы с начальным положением $(10, 20)$, скоростью $(0, -5)$, временем жизни 90 кадров, затухающую, зеленого цвета необходим следующий вызов функции:

```
Start_Particle(PARTICLE_TYPE_FADE,
          PARTICLE_COLOR_GREEN,
          90,
          10, 20, 0, -5);
```

Конечно, на систему частиц действуют гравитация и ветер, так что вы должны определить их глобально. В системе без ветра и с небольшой гравитацией это будет выглядеть следующим образом:

```
particle_gravity = 0.1;
particle_wind    = 0.0;
```

Теперь пришло время решить, как же мы должны обрабатывать движение частиц. Должны ли, например, они, пересекая границы экрана, исчезать или появляться у противоположной границы? Все это зависит от типа и конкретных условий создаваемой игры. Здесь я считаю, что вышедшая за пределы экрана частица должна быть уничтожена. Итак, с учетом всего сказанного, вот как выглядит функция обработки движения частиц:

```
void Process_Particles(void)
{
// Перемещение и анимация частиц

for (int index=0; index<MAX_PARTICLES; index++)
{
  // Активна ли данная частица?
  if (particles[index].state == PARTICLE_STATE_ALIVE)
  {
    // Перенос частицы
    particles[index].x+=particles[index].xv;
    particles[index].y+=particles[index].yv;

    // Обновление скорости с учетом гравитации и ветра
    particles[index].xv+=particle_wind;
    particles[index].yv+=particle_gravity;

    // Анимация на основе типа частицы
    if (particles[index].type==PARTICLE_TYPE_FLICKER)
    {
      // Выбор и назначение цвета
      particles[index].curr_color =
           RAND_RANGE(particles[index].start_color,
                particles[index].end_color);

      // Обновляем счетчик
      if (++particles[index].counter >=
          particles[index].max_count)
      {
        // Уничтожаем частицу
        particles[index].state =
          PARTICLE_STATE_DEAD;
      } // if
    } // if
    else
    {
      // Обновляем цвет
      if (++particles[index].counter >=
          particles[index].max_count)
      {
        // Сброс счетчика
        particles[index].counter = 0;

        // Обновление цвета
        if (++particles[index].curr_color>
            particles[index].end_color)
        {
```

```
            // Уничтожение частицы
            particles[index].state =
              PARTICLE_STATE_DEAD;
          } // if
        } // if
      } // else
    } // if
} // for index
} // Process_Particles
```

Надеюсь, принцип работы этой функции совершенно понятен. Каждая частица перемещается, обновляются ее координаты, скорость с учетом действующих сил, выполняется проверка выхода частицы за экран — и это все. Теперь нужно вывести частицы на экран. Это можно выполнить множеством способов, но я просто хочу вывести их изображения в буфер при помощи следующей функции:

```
void Draw_Particles(void)
{
// Вывод частиц на экран

// Блокируем заднюю поверхность
DDraw_Lock_Back_Surface();

for (int index=0; index<MAX_PARTICLES; index++)
{
  // Проверка активности частицы
  if (particles[index].state==PARTICLE_STATE_ALIVE)
    {
    // Получение координат частицы
    int x = particles[index].x;
    int y = particles[index].y;

    // Ограничение экраном
    if (x >= SCREEN_WIDTH || x < 0 || y >= SCREEN_HEIGHT || y < 0)
      continue;

    // Вывод
    Draw_Pixel(x,y,particles[index].curr_color, back_buffer, back_lpitch);

  } // if

} // for index

// Разблокирование вторичной поверхности
DDraw_Unlock_Back_Surface();

} // Draw_Particles
```

Вот практически и все. Нам еще нужна функция, которая бы создавала эффект взрыва, следов реактивной струи и т.п.

Генерация начальных условий

Это интересный раздел — здесь вам придется напрячь свое воображение. Начнем с алгоритма создания реактивного следа, который представляет собой не что иное, как

частицы, испускаемые из точки (emit_x, emit_y) с несколько различающимися временами жизни и начальным положением. Вот возможная реализация такого алгоритма:

```
// Испускание частиц с вероятностью 1/10
if ((rand()%10)==1)
  Start_Particle(PARTICLE_TYPE_FADE,
          PARTICLE_COLOR_GREEN,
          RAND_RANGE(90,150),
          emit_x + RAND_RANGE(-4,4),
          emit_y + RAND_RANGE(-4,4),
          RAND_RANGE(-2,2),
          RAND_RANGE(-2,2));
```

При перемещении источника частиц, разумеется, изменяются его координаты emit_x и emit_y. Для большей реалистичности вы должны учитывать, что скорость испускаемой частицы в нашей системе координат представляет собой векторную сумму скорости испускания (относительно источника) и скорости самого источника, т.е. код должен выглядеть примерно так:

```
// Испускание частиц с вероятностью 1/10
if ((rand()%10)==1)
  Start_Particle(PARTICLE_TYPE_FADE,
          PARTICLE_COLOR_GREEN,
          RAND_RANGE(90,150),
          emit_x + RAND_RANGE(-4,4),
          emit_y + RAND_RANGE(-4,4),
          emit_xv + RAND_RANGE(-2,2),
          emit_yv + RAND_RANGE(-2,2));
```

А теперь попробуем смоделировать взрыв, который выглядит подобно показанному на рис. 13.43. Частицы разлетаются в произвольных направлениях со скоростями, лежащими в некоторых пределах значений.

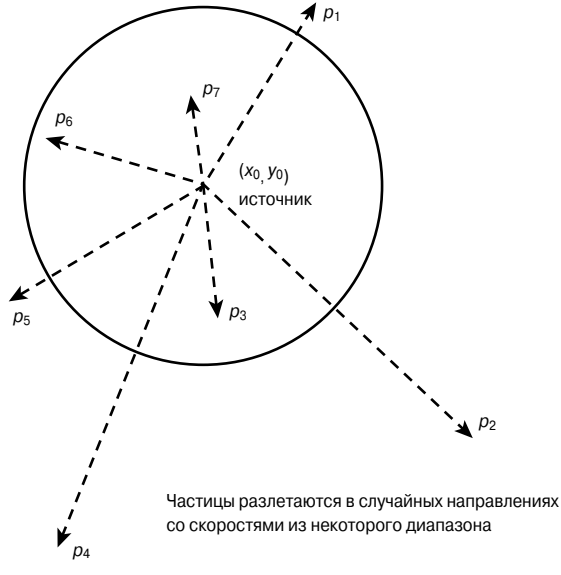


Рис. 13.43. Модель взрыва

Это очень простая модель. Все, что нужно, — это получить случайное число частиц со случайными скоростями, равномерно распределенными по направлению. Затем частицы падают под действием гравитации и либо вылетают за экран, либо прекращают существование по завершении времени жизни. Вот пример кода для генерации частиц взрыва:

```
void Start_Particle_Explosion(int type, int color,
                      int count, int x, int y,
                      int xv, int yv,
                      int num_particles)
{
while(--num_particles >=0)
  {
  // Вычисляем случайный угол...
  int ang = rand()%360;

  // ...и случайную скорость
  float vel = 2+rand()%4;

  Start_Particle(type,color,count,
           x+RAND_RANGE(-4,4),y+RAND_RANGE(-4,4),
           xv+cos_look[ang]*vel,
           yv+sin_look[ang]*vel);

  } // while

} // Start_Particle_Explosion
```

Функция Start_Particle_Explosion получает в качестве параметров тип частиц, цвет, необходимое количество частиц, а также координаты и скорость источника частиц. Затем функция генерирует требующееся множество частиц.

Для создания других спецэффектов разрабатывайте собственные функции. Например, при столкновении космического корабля с метеоритом он может распасться на частицы, разлетающиеся в виде кольца. Создать такой эффект очень просто: немного измените функцию создания взрыва так, чтобы все частицы разлетались с одной и той же скоростью, но в разных направлениях.

```
void Start_Particle_Ring(int type, int color, int count,
                 int x, int y, int xv, int yv,
                 int num_particles)
{

// Случайная скорость вычисляется до цикла
float vel = 2+rand()%4;

while(--num_particles >=0)
  {
  // Вычисляем случайное направление движения
  int ang = rand()%360;
  // Создаем частицу
  Start_Particle(type,color,count,
           x,y,
           xv+cos_look[ang]*vel,
           yv+sin_look[ang]*vel);
  } // while
} // Start_Particle_Ring
```

Теперь у вас есть все необходимое для создания красивых эффектных систем частиц. В инициализирующей фазе игры вы вызываете функцию Init_Reset_Particles(), затем в основном цикле осуществляете вызов Process_Particles(), и игровой процессор сделает за вас всю остальную работу! Конечно, не забудьте в нужный момент вызвать функцию генерации системы частиц. Ну и в качестве очевидных усовершенствований вы можете несколько изменить систему управления памятью и обеспечить возможность существования любого количества частиц, а также учесть возможность столкновения частиц друг с другом и с другими объектами игры.

На прилагаемом компакт-диске находится демонстрационная программа DEMO13_10.CPP, которая использует описанные технологии.

Построение физических моделей игр

В этой главе вы получили массу различной информации из самых разных областей механики. Теперь главное — ее корректное применение, чтобы разрабатываемая игра выглядела реалистично. Моделировать реальность абсолютно точно не просто не имеет смысла, а совершенно невозможно, поэтому следует ограничиться некоторым приближением. Каким именно — подскажет сама игра. Например, в автогонках следует всерьез смоделировать в первую очередь действие сил трения, иначе будет создаваться впечатление, что ваш автомобиль едет не по дороге, а по рельсам! Если вы моделируете полет сквозь поле астероидов и игрок расстрелял крупный астероид, вряд ли так уж необходимо точно передавать движение образовавшихся осколков. Достаточно просто отправить их по некоторым траекториям, которые довольно прилично выглядят со стороны.

Структуры данных для физического моделирования

Один из наиболее часто задаваемых мне вопросов (не считая вопроса о том, как компилировать программы DirectX при помощи VC++) — это какие именно структуры данных следует использовать для физического моделирования. Но ведь никаких физических структур данных попросту не существует! Большинство физических моделей основаны на самих объектах игры; просто к структурам данных, которые описывают эти объекты, нужно добавить необходимые для работы физической модели поля — только и всего. Обычно в играх приходится иметь дело со следующими параметрами, описывающими объекты.

- Координаты и скорость объекта.
- Угловая скорость объекта.
- Масса, коэффициент трения и другие физические свойства объекта.
- Геометрия объекта для работы физической модели (это просто те геометрические размеры, которые используются для физических вычислений; вместо реальной геометрии объекта вы можете использовать сферы, прямоугольники и другие упрощенные формы).
- Внешние глобальные силы: гравитация, ветер и т.п.

Моделирование на основе кадров и на основе времени

Это последний вопрос, который я хотел бы затронуть в связи с тем, что он приобретает все большую важность в трехмерных играх. Как уже неоднократно отмечалось, в игре используется цикл, схематично представленный на рис. 13.44. Мы предполагаем, что игра работает с постоянной частотой кадров R. Если эта скорость не выдерживается, ничего страшного: игра у нас просто окажется несколько замедленной. Но что, если вы не

хотите замедлять игру? Что, если корабль должен добраться из пункта A в пункт B ровно за 10 секунд, независимо от того, какая частота вывода кадров достигается в игре? В этом случае используется *моделирование на основе времени.*

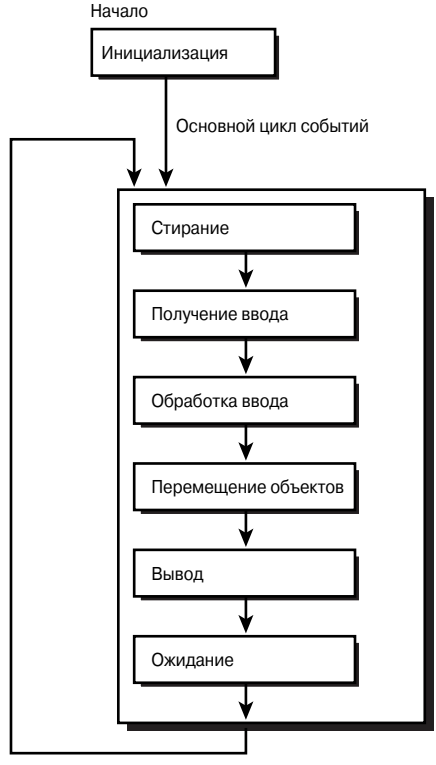


Рис. 13.44. Игровой цикл

Моделирование на основе времени отличается от моделирования на основе кадров тем, что во всех кинематических уравнениях, описывающих перемещения объекта, используется реальное время *t*. Например, при моделировании на основе кадров там, где у нас используется код

```
x = 0, y = 0;

x += dx;
Y += dy;
```

и игра осуществляет вывод со скоростью 30 fps, за 30 кадров или, что то же, за 1 секунду значения x и y станут равны

```
x = 30*dx;
y = 30*dy;
```

Если dx = 1 и dy = 0, то объект переместится на 30 пикселей в направлении x. Все хорошо, если вы можете гарантировать постоянную скорость вывода кадров. Но что, если она упадет до 10 fps? Тогда за ту же секунду вы получите

```
x = 10*dx = 10;
y = 10*dy =  0;
```

Такое замедление игры вряд ли приемлемо. Кроме того, это совершенно недопустимо в сетевой игре. В этом случае вам придется либо использовать моделирование на основе кадров и синхронизировать все машины по самой слабой, либо перейти к моделированию на основе времени, что гораздо более реалистично. Да и почему я должен страдать из-за того, что кто-то никак не соберется сменить свой старенький 486-й на такой же, как у меня, Pentium IV 2.4 GHz?

Для реализации движения и кинематики на основе реального времени вы должны использовать его во всех уравнениях, т.е., например, при перемещении объекта нужно выяснить, сколько времени прошло с момента последнего перемещения, подставить его в ваши уравнения и вычислить необходимые изменения координат и скоростей. Тогда даже при уменьшении скорости вывода игры скорость движения объектов не изменится, так как они не будут привязаны к частоте кадров, и перемещение объектов между двумя кадрами попросту увеличится. Вот как схематично должен выглядеть основной цикл игры:

```
while(1)
{
  t0 = Get_Time(); // Время в миллисекундах

  // Работа, работа, работа...

  // Перемещение объектов
  t1 = Get_Time();
  Move_Objects(t1 - t0);

  // Вывод изображения
  Render();

} // while
```

Здесь для перемещения объектов используется интервал реального времени. Считая, что скорость вывода кадров — 30 fps, получаем, что скорость перемещения объекта из предыдущего примера составляет dx = 30 пикселей/1 секунду = 0.03 пикселя в миллисекунду. Следовательно, за время t_1-t_0 изменение координаты x объекта составит $0.03(t_1-t_0)$:

```
x = x + 0.03*(t1 - t0);
```

Если частота кадров упадет до 1 fps, то интервал t_1-t_0 станет равным 1000 ms и смещение объекта за одну секунду все равно останется равным 30 пикселям.

Понятно, что при работе с такими малыми величинами вы уже не можете использовать целые числа и должны применять числа с плавающей точкой, но, думаю, сама идея вам понятна.

На прилагаемом компакт-диске находится демонстрационная программа DEMO13_11.CPP, которая осуществляет перемещение объекта с постоянной скоростью, несмотря на различную частоту кадров, которую вы можете изменять при помощи клавиш управления курсором. Обратите внимание на качество анимации при разных частотах кадров.

Резюме

В этой главе вы практически получили второе образование — физика. Надеюсь, что мои усилия по написанию этой главы не пропали даром и посеянное зерно знаний и идей прорастет всходами реализма на ниве ваших игр :-). Законы физики всеобщи, и решение задачи о столкновении двух шаров применимо в принципе и к столкновению метеорита с космическим кораблем, и в какой-нибудь игре в теннис. Главное — знание законов физики и математического аппарата для их применения; именно этому и была посвящена данная глава.

ГЛАВА 14

Генерация текста

В этой главе рассматриваются старые игры, известные как “текстовые приключения” или просто текстовые игры. Много лет назад (в конце 70-х или начале 80-х годов) компьютеры не имели таких графических возможностей, как сегодня. Из-за этого большинство игр были описательными, а не графическими. Для передачи состояния игры использовался текст; и наоборот, чтобы управлять характером игры, игрок должен был вводить простые предложения на английском языке. Одной из наиболее известных игр, созданной в начале 80-х годов, была игра под названием Zork компании Infocom. Она приобрела необычайную популярность, поскольку ее интерпретатор языка был самым современным на тот момент, а игровая среда — наиболее мощной и надежной. Кроме того, пользователь мог внести изменения почти в любое предложение и, таким образом, придумывать любые действия, на которые была способна его фантазия.

Материал, который я предлагаю на ваше рассмотрение, не слишком сложен, но отличается от всего ранее изученного. Здесь будет много новых терминов, причем некоторые из них поначалу будут не совсем понятны. Но к концу главы вы уже сможете создать собственную текстовую игру. Рассмотрим следующие темы.

- Что такое текстовая игра
- Как работает текстовая игра
- Получение входных данных из внешнего мира
- Анализ языка и грамматический разбор предложения
- Лексический анализ
- Синтаксический анализ
- Семантический анализ
- Объединение компонентов игры
- Представление мира игры

- Размещение объектов в мире игры
- Выполнение событий
- Перемещение объектов
- Система инвентаризации
- Реализация обзора, звука и запаха
- Игра в реальном времени
- Обработка ошибок
- Все об игре “Земля теней”
- Создание “Земли теней” и игра в нее
- Цикл игры “Земля теней”
- Победитель игры

Что такое текстовая игра

Текстовая игра — это видеоигра без видео! Или, по крайней мере, без графики. Текстовые игры похожи на интерактивные книги, которые вы читаете в ходе игры. Пользователь включает собственное воображение для придумывания сюжета. Вероятно, вы, как дитя эпохи графического интерфейса, даже никогда не видели текстовую игру. Интерфейс в применении к текстовой игре означает примерно следующее:

Что вы хотите сделать? — Съесть яблоко.
М-мм, как вкусно!
Что вы хотите сделать? —

Я показал вам короткий диалог между компьютером и игроком. В большинстве текстовых игр компьютер просит игрока сообщить о своих намерениях. Затем компьютер разбивает предложение на компоненты и смотрит, является ли предлагаемое действие разрешенным, и если да, то совершает его.

Важно помнить, что в текстовой игре нет ни изображения, ни звука. Создаваемый образ находится только в голове игрока. А поскольку этот образ создается при чтении описания, используемый для этого язык должен быть по возможности легким и поэтичным. Например, вы разрабатываете текстовую игру, где есть ванная комната. Когда игрок просит сказать, на что это похоже, то одно из возможных описаний может выглядеть так:

...вы видите белую ванную с висящими на вешалках полотенцами...

Здесь все верно, но слишком скучно. Лучше сказать так:

...Вас поразил размер ванной комнаты. Слева вы видите большую душевую кабину из розового стекла, вход в которую отделан маленькими гладкими камешками. Справа находится мраморный бассейн с серебряными кранами. Сверху через три застекленных окна на вас льется солнечный свет. Пол аккуратно выложен черными и белыми кафельными плитками...

Очевидно, что вторая версия значительно лучше. Она создает в вашей голове образ, а это и есть ключ к приключению в тексте. Несмотря на то что интерфейсом в текстовой игре является просто текст, увлекательность игры ограничена только фантазией ее создателя и игрока. Обычно для создания мира игры в ее базе данных содержатся сотни страниц описательного текста, выводимого в ответ на запрос игрока.

Технология текстовых игр основана на технологиях компиляции в сочетании с изящными структурами данных и алгоритмами. Помните: компьютеры не понимают естественные языки, и заставить их понять, что такое существительное, глагол, прилагательное

и т.д. — сложная задача (жаль, что я не был достаточно внимателен, когда мои учителя показывали мне, как схематически построить предложение). Многие считают, что создать текстовую игру очень легко, но они глубоко заблуждаются.

Лично я полагаю, что те, кто пишет текстовые игры, знают об информационных технологиях, структурах данных и математике гораздо больше, чем программисты, создающие обычные приключенческие игры.

Специалистам текстовых игр хорошо знакомы технологии компиляторов и интерпретаторов, а это довольно сложные вещи. Даже сегодня текстовые игры не потеряли свое значение, однако теперь они дополнены потрясающей графикой. Ролевые игры (Role Playing Games — RPG), т.е. те, в которых игрок исполняет определенную роль, фактически эволюционировали из текстовых игр. Многие RPG-игры имеют текстовые интерфейсы, так что игрок может вести диалог и задавать вопросы персонажам игры.

Я не собираюсь читать вам курс по разработке компиляторов, но приведу некоторые базовые понятия и методы, используемые для создания текстовой игры. А в конце покажу настоящую текстовую игру под названием "Земля теней".

Как работает текстовая игра

Это хороший вопрос. На него есть много ответов, но все они верны только до некоторой степени, хотя и имеют много общего. Во-первых, интерфейс игры двунаправленный и, кроме того, чисто текстовый. Это означает, что компьютер может выводить информацию для игрока только в виде текста и игрок может вводить данные тоже только в виде текста. Кроме того, в игре нет джойстика, мыши или, например, световых перьев. Во-вторых, игра состоит из некоторого типа "миров" (сцен), будь то прерии, здание или космическая станция. Миры же состоят из геометрии, описаний и правил. Геометрия представляет собой фактическую геометрию мира, размер и размещение комнат, коридоров, водоемов и т.д. Описание является текстом, который служит для того, чтобы игрок мог представить себе, как выглядит, например, местность, какими звуками и запахами она наполнена. Правила игры определяют, что можно, а что нельзя делать. Например, вы не можете есть скалы, но можете съесть сэндвич.

Когда геометрия, описания и правила игры определены, структуры данных, алгоритмы и программное обеспечение позволяют игроку взаимодействовать с окружающим миром, который необходимо создать. Главная часть программного обеспечения — синтаксический анализатор входных данных, который отвечает за перевод и осмысление вводимого игроком текста. Игрок общается с игрой с помощью обычных английских слов, из которых можно создавать развернутые предложения, небольшие фразы и отдельные короткие команды. Синтаксический анализатор и все его компоненты разбивают предложение на отдельные слова, анализируют смысл предложения и затем выполняют соответствующие функции, чтобы создать ситуацию, заказанную игроком.

Однако здесь кроется и подводный камень. Заставить компьютер понимать естественный язык — задача, которая тянет, как минимум, на докторскую степень. Как программист игр, вы должны дать игроку возможность использовать некоторое подмножество естественного языка и ограничиться несколькими простыми правилами построения предложений. Например, вскоре я познакомлю вас с игрой под названием "Земля теней", которая имеет очень небольшой словарь, представленный в табл. 14.1[1].

[1] Там, где это возможно и не искажает замысел автора, мы старались перевести английский текст на русский язык. Однако многое перевести просто невозможно в силу отличия правил английской и русской грамматики. Поэтому язык рассматриваемой в главе игры остался английским. — *Прим. ред.*

Таблица 14.1. Словарь игры "Земля теней"

Слово	*Перевод*	*Используется как*
LAMP	Лампа	Существительное
SANDWICH	Сэндвич	Существительное
KEYS	Ключи	Существительное
EAST	Запад	Существительное
WEST	Восток	Существительное
NORTH	Север	Существительное
SOUTH	Юг	Существительное
FORWARD	Вперед	Наречие
BACKWARD	Назад	Наречие
RIGHT	Вправо	Наречие
LEFT	Влево	Наречие
MOVE	Двигаться	Глагол
TURN	Повернуться	Глагол
SMELL	Пахнуть	Глагол
LOOK	Смотреть	Глагол
LISTEN	Слушать	Глагол
PUT	Положить	Глагол
GET	Взять	Глагол
EAT	Есть	Глагол
INVENTORY	Делать запасы	Глагол с существительным
WHERE	Где, куда	Наречие
EXIT	Выход	Существительное
THE		Артикль
IN	В	Предлог
ON	На	Предлог
TO	К	Предлог
DOWN	Вниз	Наречие

Словарь игры "Земля теней" очень маленький, но вы будете поражены, узнав, как много предложений можно построить на его основе. Смысл всем этим конструкциям придает использование некоторого общего алгоритма. Данный процесс называется ***синтаксическим анализом*** (syntactical analysis), он сложен и утомителен. Синтаксический анализ рассматривается во многих книгах, среди которых следует упомянуть знаменитую книгу Ахо, Сети и Ульмана "Компиляторы"[2]. Чтобы не усложнять себе жизнь, ограничимся несколькими грамматическими правилами в добавление к словарю, что и позво-

[2] Ахо А., Сети Р., Ульман Д. *Компиляторы: принципы, технологии и инструменты.* — М.: Издательский дом "Вильямс", 2001. — *Прим. ред.*

лит нам создать небольшой “язык”. Пользователь должен строить свои предложения в соответствии с правилами этого языка.

Как только пользователь набрал какой-то текст и компьютер понял, что он попытался сказать, он продолжит выполнение игры с учетом ввода пользователя и выдаст ему результат выполнения запрошенных действий. Например, если игрок введет “look” (смотри), то игра обратится к базе данных текущей сцены и выведет описание помещения с учетом позиции игрока. Если в игре есть подвижные объекты, то имеется и вторая фаза описания: логика игры проверит наличие каких-либо подвижных объектов в поле зрения персонажа, а затем напечатает их описания. Однако конечное описание должно казаться единым, а не состоящим из отдельных несвязанных частей. Например, программа вначале должна проверить, есть ли в помещении какие-либо объекты, и если да, то отметить их и слегка изменить последнее предложение в статическом описании помещения с тем, чтобы в результате получилось одно предложение, связывающее два отдельных предложения посредством союза “и” или “а”, и чтобы объекты при перечислении не появлялись из ниоткуда.

Например, комната может иметь такое статическое описание:

...Вы очутились в комнате с высокими стенами, на которых развешаны полотна времен Римской империи.

А теперь давайте добавим, что в комнате есть дополнительные объекты, например цветок в горшке. Тогда компьютер должен выдать ответ примерно такого плана:

...Вас окружают высокие стены, на которых развешаны полотна времен Римской империи. В восточном углу комнаты находится цветок в горшке.

Оптимальный алгоритм для печати этого описания должен изначально учитывать наличие в комнате объекта, и описание должно иметь примерно такой вид:

...Вас окружают высокие стены, на которых развешаны полотна времен Римской империи, а в восточном углу комнаты находится цветок в горшке.

Хотя такое предложение немного искусственно и примитивно, оно все же лучше, чем два отдельных предложения. Вы должны так строить выходные предложения, чтобы не казалось, что они взяты из статической базы данных.

Конечно, здесь требуется больше, нежели простой грамматический разбор текста. Игра должна иметь определенные структуры данных для представления мира и объектов внутри этого мира. Это представление, каким бы оно ни было, должно иметь достоверный геометрический вид, и если игрок перемещается, то он должен быть уверен, что найдет ключ там, где он его ищет. Это означает, что игровой мир должен быть представлен в виде карты, по которой может перемещаться игрок. Представление данных игры должно быть геометрически связано с виртуальным игровым пространством. Игра должна также иметь некоторую структуру данных, описывающую состояние игрока (в это понятие входит состояние его здоровья, позиция на карте, наличие инвентаря и прочие характеристики).

И наконец, должны быть реализованы и другие аспекты игры, такие, как цель и враги (если они есть). Цель может быть простой, например нужно найти волшебный кубок и доставить его в определенное место. Но цель может быть и сложной, состоящей в решении нескольких головоломок; при этом происходит диалог между игроком и каким-либо персонажем игры. Персонажи текстовой игры реализуются как структуры данных, без видеовывода на экран; однако эти персонажи могут перемещаться по игровому миру, передвигать другие объекты, принимать пищу, бороться с игроком... Все эти аспекты текстовой игры должны быть реализованы таким образом, чтобы создавалась полная иллюзия реального мира. Вы должны понимать, что, хотя игрок и не может видеть игру, он хочет ее ощущать.

Повторим еще раз все компоненты текстовой игры, изучив схему на рис.14.1.

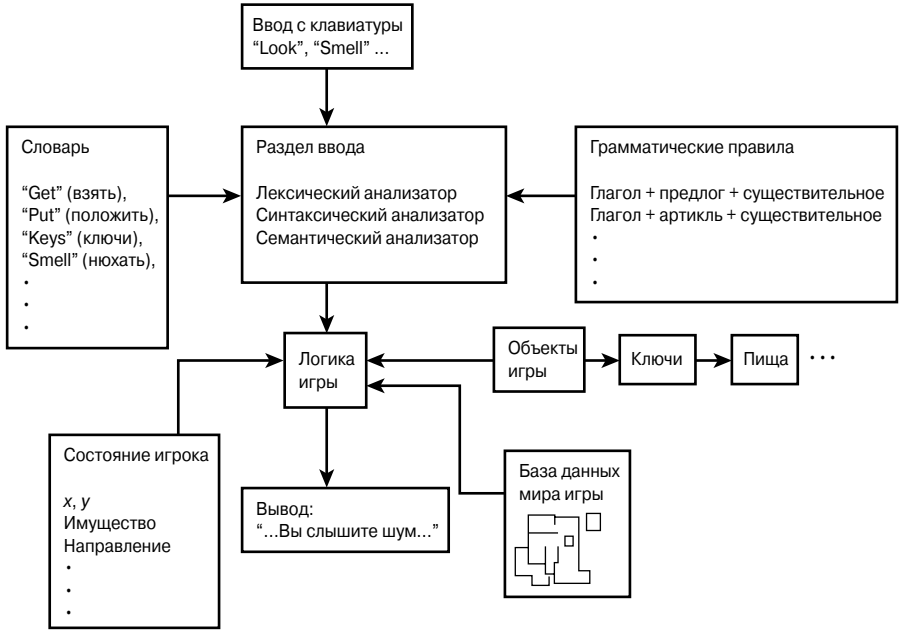


Рис. 14.1. Компоненты текстовой игры

Из схемы на рис. 14.1 видно, что программа включает вводную часть, которая осуществляет грамматический разбор вводимых игроком команд и затем пытается исполнить эти команды. Синтаксический анализатор понимает слова только из определенного словаря; кроме того, предложения, создаваемые с помощью словаря, ограничены правилами языка, разработанного автором игры. Эта схема содержит также набор структур данных, хранящих представления о мире игры, положение объектов, описание строк для показа внешнего вида объектов, “воспроизведения” звуков и запахов. Наряду с этим существует представление самого игрока и его имущества, а также представление его врагов. И наконец, в программе есть множество функций, правил и небольших деталей, которые и заставляют ее работать. Рассмотрим некоторые из них.

Получение входных данных из внешнего мира

Поскольку текстовые игры в качестве единственного инструмента для ввода данных используют клавиатуру, этот процесс следует по возможности облегчить. Игрок набирает на клавиатуре предложения, которые являются командами, заставляющими игру выполнить какое-либо действие. На первый взгляд это достаточно просто, но проблема в том, как прочитать напечатанное предложение. Стандартная функция `scanf()` работает некорректно из-за возможных пробелов и множества аргументов. Нам необходима функция, которая будет возвращать строку, независимо от составляющих ее символов. Ввод данных должен быть остановлен тогда и только тогда, когда пользователь нажмет <Enter>. С этого момента предложение перейдет к следующей части кода в цепи грамматического разбора предложения. Таким образом, задача состоит в получении одной строки текста

без использования редактора. Решение задачи — в использовании посимвольного ввода и построении строки до тех пор, пока не будет нажата клавиша <Enter>, что осуществляется при помощи функции getch() с проверкой введения символа забоя и символа возврата каретки. В листинге 14.1 представлена типичная функция ввода строки.

Листинг 14.1. Функция однострочного ввода с возможностью редактирования

```
char *Get_Line(char *buffer)
{
// Эта функция дает одну строку ввода
// и допускает наличие в ней пробелов
int c, index=0;
// Цикл, пока пользователь не нажмет <Enter>
while ((c=getch())!=13)
{
// Реализация забоя
   if (c==8 && index>0)
   {
      buffer[--index] = ' ';
      printf("%c %c",8,8);
   } // if
   else if (c>=32 && c<=122) // Только английские буквы
   {
      buffer[index++] = c;
      printf("%c", c);
   } // if
} // while

// Завершаем строку
buffer[index] = 0;
// Вернуть указатель на buffer или NULL
if (strlen(buffer)==0)
   return(NULL);
else
   return(buffer);
} // Get_Line
```

Функция получает в качестве параметра указатель на буфер, в котором после выполнения функции будет размещена введенная строка. Во время выполнения функция Get_Line() позволяет пользователю ввести строку текста и отредактировать ошибки с помощью клавиши забоя. При введении символов они будут отражаться на экране, так что пользователь сможет увидеть то, что он напечатал. (В некоторых компиляторах (например, Watcom) после вывода на экран следует сбросить буфер вызовом fflush(stdout). — *Прим. ред.*) Когда пользователь нажмет клавишу <Enter>, в буфер будет добавлен нулевой символ и работа функции будет на этом завершена.

После введения пользователем строки текста следует приступить к ее разбору. Процесс синтаксического разбора предложений состоит из многих стадий, но, прежде чем мы сможем выполнить разбор предложения и понять, что оно должно означать, необходимо познакомиться с тем, что собой представляет язык и как он построен.

Анализ и синтаксический разбор языка

Прежде чем пользователь что-то введет, необходимо определить “язык”, который он при этом использует. Язык состоит из словаря и набора правил (грамматики). В тексто-

вой игре используется английский язык. Это означает, что в словарь должны входить английские слова и правила английской грамматики. Выполнение первого требования не вызывает затруднений. Удовлетворить второму требованию не так-то просто. На мой взгляд (хотя многие учителя английского языка и не разделяют мою точку зрения), структура английского языка является достаточно сложной. Он полон противоречий, разных точек зрения по одному и тому же вопросу и способам построения предложения и т.д. Английский язык не такой однозначный и точный, как любой из языков программирования, а это означает, что, как программисты текстовых игр, мы можем разрешить игроку использовать только небольшой набор возможных грамматических конструкций, которые можно создать с помощью предоставляемого словаря.

Обычно с этим нет проблем. Игрок очень быстро привыкает к английскому "новоязу" и чувствует себя довольно комфортно и уверенно среди лаконичных и законченных предложений. Итак, первое, что необходимо сделать программисту, — это составить словарь, причем способ составления должен предусматривать возможность его продолжения. Дело в том, что, когда игра уже написана, разработчику может понадобиться добавить в нее новый объект — какой-нибудь "zot", и тогда это слово нужно будет ввести в словарь, чтобы игрок впоследствии мог его использовать. Во-вторых, разработчик может решить, что для разнообразия предложений необходимо иметь больше предлогов. Например, чтобы уронить объект игрок может напечатать такую фразу:

drop the keys (уронить ключи).

Здесь все предельно ясно. Слово "drop" (уронить) — глагол, "the" — артикль, и "keys" — существительное. Однако для игрока более естественно напечатать следующее:

drop down the keys

или

drop the keys down.

Оба варианта одинаково хороши, все зависит от того, как именно привык разговаривать конкретный человек — именно это и определяет стиль общения. Поэтому в словаре должны быть слова для всех задействованных в игре объектов (существительных) наряду с небольшим количеством ***предлогов***. В качестве эмпирического правила запомните, что лучше внести в словарь предлоги, обозначающие направление, этого обычно вполне достаточно. В табл. 14.2 приведен перечень предлогов, которые чаще всего используют в текстовых играх.

Таблица 14.2. Некоторые наиболее распространенные предлоги

Предлог	*Перевод*
From	Из
On	На
In	В
Down	Вниз, указывает на движение вниз
Up	Вверх, указывает на движение вверх
Behind	За, сзади, позади
Into	В (внутри)
Before	Перед, до
At	У, в, за, на

Следующий класс слов, которые необходимо иметь в словаре, — это ***глаголы действия***, т.е. слова, обозначающие действие. За этими словами обычно следуют слова или фразы, описывающие, какое именно действие имеет место. Например, в текстовой игре глагол действия "move" (двигать(ся), перемещать(ся)) может использоваться для перемещения игрока внутри игрового мира. Ниже приведен ряд возможных предложений с глаголом "move".

1. Move north (двигаться на север).
2. Move to the north (двигаться к северу).
3. Move northward (двигаться в северном направлении).

Первое предложение достаточно простое. В нем говорится, что надо двигаться на север, причем для указания направления не используется никакого предлога. Во втором предложении также говорится о движении к северу, но по построению оно содержит предложную группу "to the north". И наконец, в третьем предложении для указания направления использовано прилагательное (в русском языке — причастие) "northward" (расположенный к северу). Технически можно доказать, что каждое предложение может подразумевать разные вещи, но для нас, как программистов, все они означают одно и то же. Все три предложения указывают направление на север. Следовательно, мы видим, что использование артикля "the" и предлогов абсолютно необходимо для построения предложения, которое имеет небольшое отличие от других, равноценных по смыслу предложений. (Можно привести аналогию из русского языка, когда по конструкции предложения отличаются между собой, а по смыслу — идентичны. Например: "плывите морем", "плывите по морю" и "плывите морским путем". — *Прим. перев.*)

Вернемся к глаголам действия. Игра может содержать длинный список таких глаголов, причем многие из них используются самостоятельно с неявными объектами. Допустим, в словаре игры есть глагол "smell" (нюхать). Вы можете составить такую фразу:

smell the sandwich (понюхать сэндвич).

Здесь все ясно. А вот команда

smell (понюхать)

уже не так четко выражает мысль. Конечно, понятно, какое действие игрок хочет совершить, но неясно, на что (или на кого) направлено это действие. В этот момент в игру вступает понятие контекста. Так как в последней команде прямой или непрямой объект действия "smell" отсутствует, программа предполагает, что игрок имеет в виду внешнюю среду или последний объект, над которым было совершено действие. Например, если игрок только что поднял камень и вводит команду "smell", то компьютер может переспросить: "the rock?" (камень?) и после ответа игрока "yes" на экран будет выведено соответствующее описание. С другой стороны, если игрок перед этим ничего не брал в руки, то команда "smell" может вызвать общее описание запаха комнаты, где находится игрок.

После того как программист выберет все глаголы, существительные, предлоги и артикли, которые могут использоваться в языке, он должен выбрать основные грамматические правила этого языка. Эти правила описывают возможные конструкции предложений, которые разрешены в рамках языка. Эта информация используется синтаксическим и ***семантическим*** анализаторами для установления смысла предложения и его корректности. В качестве примера давайте составим словарь и определим грамматику языка (табл. 14.3).

Таблица 14.3. Образец словаря

Слово	*Используется в качестве*	*Имя типа*
Rock (камень, скала)	Существительного	OBJECT
Food (пища)	Существительного	OBJECT
Table (стол)	Существительного	OBJECT
Key (ключ)	Существительного	OBJECT
Get (взять)	Глагол	VERB
Put (положить)	Глагол	VERB
The	Артикль	ARTICLE
On (на)	Предлог	PREP
Onto (на)	Предлог	PREP

Теперь у нас есть словарь и мы можем создать на его основе язык, определив правила построения предложений. Конечно, не все предложения могут иметь смысл, но все они должны быть грамматически правильно построены. Правила построения предложений называют синтаксисом языка или продукциями[3] (табл. 14.4).

Таблица 14.4. Синтаксис языка Glish

OBJECT	→	"rock" | "food" | "table" | "key"
VERB	→	"get" | "put"
ARTICLE	→	"the" | NULL
PREP	→	"on" | "onto" |NULL
SENTENCE	→	VERB+ARTICLE+OBJECT+PREP+ARTICLE+OBJECT

Здесь "|" означает логическое ИЛИ, "+" означает соединение, а `NULL` — пустую строку.

Если вы попытаетесь теперь построить предложение, используя продукцию SENTENCE, то обнаружите, что можно составить не так уж много предложений, причем некоторые их них не будут иметь никакого семантического смысла. Например, предложение

put the rock onto the table (положите камень на стол)

имеет вполне определенный смысл, а вот смысл предложения

get the rock onto the table (возьмите камень на стол)

уже не так ясен: возможно, подразумевается, что камень нужно разместить на столе? Здесь необходимо сделать одно из двух: либо ввести в язык дополнительные правила для того, чтобы определить значение конкретных глаголов и их конструкций, либо код игры должен проверять предложение на наличие смысла. Для того чтобы вы поняли, как можно использовать синтаксис, давайте рассмотрим несколько примеров (табл. 14.5).

Если предложение не соответствует синтаксису языка, оно является некорректным, несмотря на наличие в нем смысла. Это происходит потому, что компьютер признает законность предложения, только если его конструкция подчиняется определенным прави-

[3] Здесь и далее используется терминология из упомянутой книги *Компиляторы: принципы, технологии и инструменты*. — *Прим. ред.*

лам, введенным в компьютер. Иначе говоря, каждое предложение должно проверяться на соответствие грамматике входного языка.

Таблица 14.5. Примеры предложений и их смысл

Предложение	*Ясность предложения*	*Соответствие правилам*
Put rock	4	Да
Put the rock	4	Да
Get the key	8	Да
Put down the food	10	Нет
Примечание. Синтаксис не предусматривает наличия предлога после глагола; слова "down" в словаре нет		
Put the food on rock	7	Да
Put the rock down onto table	7	Нет
Примечание. Синтаксис не предусматривает наличия двух последовательных предлогов.		

После того как предложение будет разбито на части и установлена его корректность, выполняется семантическая проверка предложения. На этой стадии проверяется наличие в нем смысла.

Например, предложение

get the key onto the table (возьмите ключ на стол)

корректно с точки зрения синтаксиса, но некорректно семантически. Вероятно, его следует пометить как некорректное, а пользователя попросить выразить свою мысль как-то иначе.

Далее возникает вопрос: "Что нужно сделать, чтобы программа выполнила то, о чем говорится в семантически корректном предложении?" Ответ связан с ***функциями действия***, названными так по аналогии с соответствующими глаголами действия в предложении. Такие функции действия могут использоваться как программы синтаксического контроля, семантического контроля или комбинации обеих программ либо как-то иначе (это зависит от вас). Но главное то, что функция действия должна вызвать выполнение какого-либо действия, причем конкретная функция отвечает за конкретное действие. Функции действия сами определяют, к какому объекту (объектам) будет применен глагол действия, после чего они выполняют запрашиваемое действие. Например, функция действия для глагола "get" (получать) проходит через следующий набор операций:

- выделение "объекта" (существительного) из предложения;
- определение, находится ли объект, которому адресовано действие, в области досягаемости игрока;
- если объект находится в пределах досягаемости, то с ним выполняется необходимое действие; кроме того, происходит обновление базы данных игры и данных об имуществе игрока.

Первый шаг (выделение объекта) выполняется с помощью исключения слов, которые не изменяют смысла предложения. Например, предлоги и артикли в нашем языке Glish не очень изменяют смысл любого отдельного предложения. Обратите внимание, что из фразы

put the key (положите ключ)

можно убрать артикль "the" и смысл предложения при этом не изменится. Исключение из предложения таких "бесполезных" слов осуществляется прозрачно для игрока, так что

он может использовать предложения, которые, на его взгляд, более корректны. Например, хотя для нашего синтаксического анализатора предпочтительнее использовать фразу

put key on table (положите ключ на стол),

игрок может предпочесть воспользоваться более "естественной", с его точки зрения, фразой

put the key onto the table

(поскольку, по его мнению, она четче формулирует мысль благодаря использованию предлога, усиливающего направление действия ("onto" вместо "on"), и определенного артикля "the"). (В русском языке для усиления направленности действия можно было бы использовать фразу "положите ключ прямо на стол". — *Прим. перев.*)

В любом случае функция действия будет отслеживать логику предложения. Как только функция действия поймет, что от нее требуется, она легко это выполнит. В нашем примере объект необходимо взять из базы данных игры и перенести в список имущества, принадлежащего игроку (который определен как массив символов или структур, перечисляющий объекты, которые хранятся у игрока). В рассматриваемой нами игре "Земля теней" данные об игроке и его имуществе содержатся в структуре, приведенной в листинге 14.2.

Листинг 14.2. Структура для хранения информации об игроке в игре "Земля теней"

```
// Эта структура хранит всю информацию об игроке
typedef struct player_typ
{
   char name [16];       // Имя игрока
   int x,y;              // Позиция игрока
   int direction;        // Направление игрока: восток,
                         // запад, север, юг
   char inventory[8];    // Объекты, принадлежащие игроку
                         // (8 карманов)
   int num_objects;      // Количество объектов у игрока
}player, *player_ptr;
```

Инвентарь в данном случае — это массив символов, представляющих объекты, которые имеет игрок. Например, 's' обозначает сэндвич (sandwich). Эта структура данных — просто пример; все можно организовать и по-другому.

По существу, программа текстовой игры должна сделать следующее: разобрать предложение, вычленить из него запрашиваемое действие, вызвать соответствующую функцию(и) и выполнить необходимые действия. Выполнение действий осуществляется путем обновления структур данных, изменения координат и вывода текста. Основная проблема в текстовой игре — это перевод вводимой командной строки в понятную компьютеру информацию, например в числа. Этот процесс называют ***лексическим анализом***.

Лексический анализ

Когда игрок вводит предложение, используя имеющийся в игре словарь, оно должно быть преобразовано из строк в "токены" (целые числа) с тем, чтобы можно было выполнить его синтаксический и семантический анализ. Причина, требующая выполнения лексического анализа, состоит в том, что работа со строками гораздо сложнее и требует больше времени и памяти, чем работа с токенами, которые обычно являются целыми числами. Поэтому одна из функций лексического анализатора преобразует вводимое предложение из строки в набор токенов. Трансляция состоит из трех частей. Первый

этап лексического анализа заключается в простом выделении и извлечении слов из предложения. Например, предложение

This is a test (это тест)

в английском варианте состоит из четырех слов.

1. This
2. is
3. a
4. test

Необходимо также учитывать точки, которые отделяют предложения друг от друга. Если пользователь собирается ввести за один раз только одну строку, то он может ставить или не ставить точку в конце вводимого предложения. Однако если игрок может использовать несколько команд одновременно, то требуется обязательный символ для разделения этих команд.

Далее нам необходимо определить способ, с помощью которого в предложении выделяют "слова", разделенные пробелами. Для этого в библиотеке языка C есть строковая функция под названием strtok(), но в некоторых компиляторах она работает некорректно, так что мы изобретем колесо и разработаем свою функцию для извлечения слов из строки.

Эту функцию несложно написать; рассмотрите, например реализацию логики выделения токенов в листинге 14.3.

Листинг 14.3. Функция выделения токенов из введенного предложения

```
int Get_Token(char *input, char *output, int *current_pos)
{
int index,    // Индекс цикла
   start,     // Точки начала токена
   end;       // Точки конца токена

// установите текущие позиции
index=start=end=*current_pos;
//
while(isspace(input[index])) || inspunct(input[index]))
{
   index++;
} // while

// Проверка конца строки
if (input[index]==NULL)
{
   // Ничего не выделено
   strcpy(output, "");
   return(0)
}// Токенов больше нет

// Начинаем поиск токена
start = index; // Пометили начало
end = index;

// Ищем конец токена
```

```
while(!isspace(input[end]) &&
    !ispunct(input[end]) &&
    input[end]!=NULL)
{
   end++;
}// while

// Создаем выходную строку
for (index=start; index<end; index++)
{
   output[index-start] = toupper(input[index]);
} // for

// Добавляем нулевой символ
output[index-start] = 0;

// Обновляем текущую позицию строки
*current_pos = end;

return(end);
} // Get_Token
```

Функция получает три входных параметра: введенную строку, из которой извлекается токен, выходную строку, в которую копируется токен, и текущую позицию в строке, которая обрабатывается. Давайте посмотрим, как выполняется синтаксический анализ с помощью этой функции. Вначале объявим пару переменных.

```
int position=0;    // используется как индекс для отслеживания
                   // текущей позиции строки
char output[16];  // выходная строка

// Начнем извлечение токенов
Get_Token("This is a test", output, &position);
```

После этого вызова переменная output должна содержать слово “This”, position будет равна 4. Давайте сделаем еще один вызов:

```
Get_Token("This is a test", output, &position);
```

Теперь переменная output должна содержать слово “is”, а обновленная переменная position будет равна 7. Если вызовы Get_Token() будут повторяться, то поочередно будет выделено и помещено в буфер каждое слово в предложении (конечно, каждая новая строка, размещаемая в output, будет затирать находящуюся там старую строку). Таким образом, у нас есть метод получения слов, из которых построено предложение. Следующая задача состоит в преобразовании этих строк в числовые символы, поскольку с ними легче работать. Задачу решают с помощью функции, в которой есть таблица строк и символов для каждого слова из словаря. Функция осуществляет поиск каждого слова из предложения и, если соответствующая строка найдена, преобразует ее в число (рис. 14.2).

Во время этой фазы лексического анализа происходит проверка наличия слов в словаре. Если слово отсутствует в словаре, то его нет и в языке, а следовательно, его использование некорректно. В качестве примера обратимся вновь к игре “Земля теней”, в которой для хранения слов из словаря игры используется структура данных, показанная в листинге 14.4.

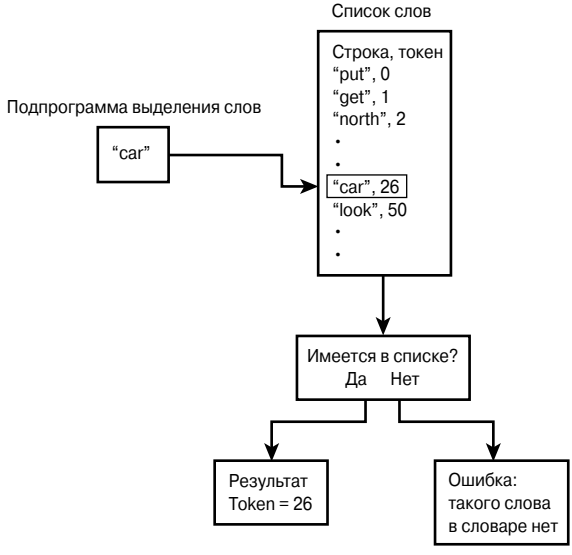


Рис. 14.2. Функция проверяет, содержится ли использованное слово в словаре игры

Листинг 14.4. Структура данных для токенов в игре “Земля теней”

```
// Структура для одиночного токена
typedef struct token_typ
{
   char symbol[16];  // Строка, представляющая токен
   int  value;       // Целое значение токена
} token, *token_ptr;
```

В этой структуре есть место как для строки, так и для соответствующего ей значения. Строка представляет собой слово из словаря, а значение может быть любым (однако эти значения должны быть взаимоисключающими, если только вы не пользуетесь синонимами). Используя описанную структуру и некоторые определения, вы можете создать очень компактную словарную таблицу, которую можно использовать в лексическом анализе для превращения слов в символы и проверки их корректности. Листинг 14.5 представляет собой таблицу-словарь, используемую в игре “Земля теней”.

Листинг 14.5. Статический словарь игры “Земля теней”

```
// Полный словарь языка “Земли теней”
token language[MAX_TOKENS] = {
  {"LAMP",    OBJECT_LAMP    },
  {"SANDWICH", OBJECT_SANDWICH },
  {"KEYS",    OBJECT_KEYW    },
  {"EAST",    DIR_1_EAST     },
  {"WEST",    DIR_1_WEST     },
  {"NORTH",   DIR_1_NORTH    },
  {"SOUTH",   DIR_1_SOUTH    },
  {"FORWARD", DIR_2_FORWARD  },
  {"BACKWARD", DIR_2_BACKWARD },
```

```
  {"RIGHT",   DIR_2_RIGHT   },
  {"LEFT",    DIR_2_LEFT    },
  {"MOVE",    ACTION_MOVE   },
  {"TURN",    ACTION_TURN   },
  {"SMELL",   ACTION_SMELL  },
  {"LOOK",    ACTION_LOOK   },
  {"LISTEN",  ACTION_LISTEN },
  {"PUT",     ACTION_PUT    },
  {"GET",     ACTION_GET    },
  {"EAT",     ACTION_EAT    },
  {"INVENTORY",ACTION_INVENTORY},
  {"WHERE",   ACTION_WHERE  },
  {"EXIT",    ACTION_EXIT   },
  {"THE",     ART_THE       },
  {"IN",      PREP_IN       },
  {"ON",      PREP_ON       },
  {"TO",      PREP_TO       },
  {"DOWN",    PREP_DOWN     }
};
```

Обратите внимание, что рядом со строками расположены символьные значения, которые имеют хорошо известные приставки, указывающие части предложения, например PREP (предлог), ART (артикль) и ACTION (глагол).

Теперь, пользуясь таблицей, можно сравнивать слова предложения с элементами в таблице и преобразовывать строки токенов в целые числа. Таким образом, введенное предложение преобразуется в строку чисел, которые удобно обрабатывать. Пример такой функции разбора показан в листинге 14.6. Этот пример также взят из игры "Земля теней", но принцип действия функции не зависит от характера игры.

Листинг 14.6. Функция преобразования слов в целочисленные токены

```
int Extract_Tokens(char *string)
{
// Функция разбивает введенную строку на токены и заполняет
// ими глобальный массив для последующей работы

int curr_pos=0,      // Текущая позиция строки
   curr_token=0,     // Текущий номер токена
   found,            // Используется для отметки наличия
                     // токена в языке
   index;            // Индекс цикла

char output[16];
// Сброс количества токенов в предложении
num_tokens=0;

for (index=0; index < 8; index++)
   sentence[index]=0;

// Выделяем слова в предложении
while(Get_Token(string, output, &curr_pos))
{
   //Проверяем наличие слова в словаре
```

```
  for(index=0,found=0; index<NUM_TOKENS; index++)
  {
    // Устанавливаем соответствие
    if (strcmp(output, language[index].symbol)==0)
    {
      // Устанавливаем флаг found
      found=1;

      // Вносим символ в предложение
      sentence[curr_token++]=language[index].value;
      break;
    } // if
  }// for index

  // Проверяем, является ли токен частью языка
  if (!found)
  {
    printf("\n%s, I don't know what \"%s\" means.",
        you.name,output);

    // неудача
    return(0);

  } // if
  num_tokens++;
}// while
}// Extract_Tokens
```

Функция оперирует с введенной строкой, которая содержит команды пользователя. Сначала функция разбивает предложение на отдельные слова с помощью функции Get_Token(). Каждое слово просматривается, сравнивается со словами из словаря и в случае обнаружения преобразуется в числовую константу и вносится в предложение токенов, которое представляет собой то же введенное предложение, но в форме чисел. Если слово в словаре найдено не будет, то программа выведет сообщение об ошибке. Все очень просто.

А теперь давайте немного отвлечемся и вкратце рассмотрим пару вопросов. Первый из них — обработка ошибок. Это очень важная часть любой программы, но особое значение она приобретает в текстовой игре, где плохой ввод может повлиять и на игровую машину, и на логику игры и вообще запутать все, что можно. Поэтому в текстовой игре должно быть множество проверок на наличие ошибок.

Во-вторых, вы заметили, что я использую очень примитивную структуру данных. Возможно, связанный список или двоичное дерево были бы более элегантным решением, но для двух десятков слов делать этого не стоит. Мой опыт подсказывает, что структура данных должна соответствовать проблеме и массивы вполне подходят для решения небольших задач. Как не стоит идти на уток с крупнокалиберной винтовкой, так для решения мелких проблем не стоит использовать связанные списки, двоичные деревья, графы и т.п.

Вы уже знаете, как преобразовать предложение в токены, и теперь самое время обсудить две следующие фазы — синтаксический и семантический анализ.

Синтаксический анализ

При трансляции текстов синтаксический анализ в широком значении определяют как фазу превращения входящего потока токенов во фразы грамматики, которые можно об-

рабатывать. Поскольку мы рассматриваем довольно простые языки и словари с небольшим запасом слов, это общее определение можно упростить. В нашем контексте синтаксический анализ означает "установление смысла предложения", т.е. выделение глаголов, определение объектов, выделение предлогов, артиклей и т.д.

Этап синтаксического анализа в наших играх параллелен выполнению действий, указанных в предложении. Это вполне допустимо, и многие компиляторы и интерпретаторы работают именно таким образом — интерпретируя код "на лету". Я думаю, вы уже поняли суть синтаксического анализа, в ходе которого вызываются функции, отвечающие за все глаголы действия в языке. Затем эти функции действия обрабатывают остальные части речи во введенном предложении (которое к этому моменту представлено в виде массива токенов) и пытаются выполнить подразумеваемое предложением действие. Например, синтаксический анализатор текстовой игры вначале может выяснить, какие глаголы действия стоят в начале предложения, а затем вызвать соответствующую функцию действия, которая будет работать с остальными членами предложения.

Конечно, остальная часть предложения (кроме глаголов) может быть обработана и протестирована на корректность до вызова функции действия, но это необязательно. Лично я предпочитаю перекладывать задачу выяснения корректности предложения на функцию действия, руководствуясь принципом: каждая функция действия представляет собой объект, который работает с предложениями некоторого типа, начинающимися с конкретного глагола. У нас нет функции, которая отвечала бы за проверку синтаксиса всех возможных предложений до вызова соответствующей функции действия.

В качестве примера рассмотрите листинг 14.7, где приведена функция, которая в игре "Земля теней" вызывает функции действия, соответствующие глаголам во вводимом предложении.

Листинг 14.7. Функция, вызывающая функции действия

```
void Verb_Parser(void)
{
  // Функция разбирает предложение и исходя из глагола
  // вызывает соответствующую функцию действия

  // Какой глагол стоит на первом месте?

  switch(sentence[FIRST_WORD])
  {
    case ACTION_MOVE:
      {
        //вызов соответствующей функции
        Verb_MOVE();
      } break;
    case ACTION_TURN:
      {
        //вызов соответствующей функции
        Verb_TURN();
      } break;
    case ACTION_SMELL:
      {
        //вызов соответствующей функции
        Verb_SMELL();
      } break;
```

```
    case ACTION_LOOK:
      {
        //вызов соответствующей функции
        Verb_LOOK():
      } break;
    case ACTION_LISTEN:
      {
        //вызов соответствующей функции
        Verb_LISTEN();
      } break;
    case ACTION_PUT:
      {
        //вызов соответствующей функции
        Verb_PUT();
      } break;
    case ACTION_GET:
      {
        //вызов соответствующей функции
        Verb_GET();
      } break;
    case ACTION_EAT:
      {
        //вызов соответствующей функции
        Verb_EAT();
      } break;
    case ACTION_WHERE:
      {
        //вызов соответствующей функции
        Verb WHERE();
      } break;
    case ACTION_INVENTORY:
      {
        //вызов соответствующей функции
        Verb_INVENTORY();
      } break;
    case ACTION_EXIT:
      {
        // вызов соответствующей функции
        Verb_EXIT();
      } break;
    default:
      {
        print("\n%s, you must start a sentence "
            "with an action verb!", you.name);
      } break;
  } // switch
} // Verb_Parser
```

Эта функция на удивление проста. Она находит первое слово в предложении (которое по определению языка должно быть глаголом) и вызывает соответствующую функцию действия. Графически этот процесс можно изобразить в виде схемы, приведенной на рис. 14.3.

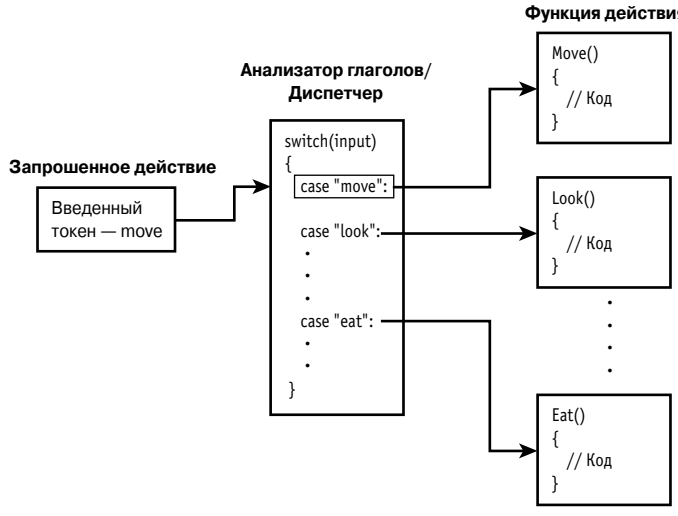


Рис. 14.3. Для каждого глагола вызывается своя функция действия

Семантический анализ

Основное количество проверок на наличие ошибок осуществляется при семантическом анализе. Однако, поскольку мы выполняем синтаксический анализ "на лету", семантический анализ осуществляется функцией действия, по сути, одновременно с синтаксическим. С помощью синтаксического анализа предложение разбивают на значимые части, а затем с помощью семантического анализа определяют, имеют ли эти значимые части предложения какой-либо смысл (я вас запутал окончательно?). В текстовых играх синтаксический и семантический анализы выполняются одновременно: пока синтаксический анализ проверяет значимость отдельных слов в предложении, семантический анализ проверяет смысл предложения.

Надеюсь, что вам все понятно. Если нет, то все должно стать понятным при рассмотрении конкретного примера.

Разработка игры

Итак, у нас есть все компоненты грамматического анализатора и интерпретатора текста. Нам осталось объединить их и построить фундамент текстовой игры (рис.14.4).

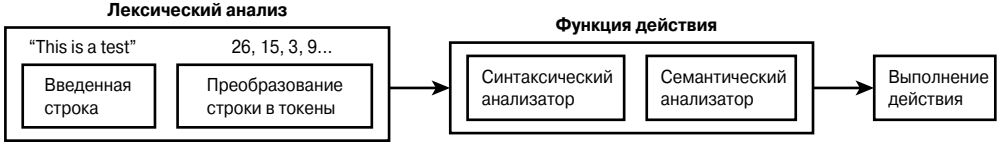


Рис. 14.4. Части интерпретатора команд

Из рисунка видно, что на первой стадии интерпретатор разбивает входной поток на слова, затем строки слов преобразуются в строки токенов. Далее поток токенов подается на вход синтаксического анализатора, передающего предложение из токенов соответствующей функции действия, которая пытается применить запрошенное игроком действие к объекту (объектам) из предложения. Одновременно с выполнением действий данная функция проводит синтаксическую и семантическую проверку.

Важно понять, что все эти процессы — разбор строки, проверка синтаксиса и т.д. — служат единственной цели: выяснить, что хочет сделать игрок. Как только цель игрока определена, ее выполнение не является сложной задачей. Не забывайте только, что все выполняемые действия направлены на внутренние структуры данных. Единственным выводом вашей программы является текст. Игрок видит мир игры только через строки статического или синтезируемого текста, который игра выводит на каждом ходу.

Надеюсь, теперь вы поняли, как работает фаза ввода текста. Уже сейчас вы можете сами создать текстовую игру, но я все же хочу рассказать вам и о других аспектах текстовых игр. Предлагаемые мною далее методики носят чисто иллюстративный характер и их не следует принимать за догму — при желании вы всегда можете найти более совершенные и удобные способы решения проблем.

Представление мира игры

Мир в текстовой игре можно представить различными способами. Но каким бы ни было это представление, оно должно иметь геометрическую взаимосвязь с виртуальной средой, внутри которой находится игрок. Например, если вы создаете трехмерную текстовую игру, то в вашей базе данных должна быть трехмерная модель. Несмотря на то что игрок не видит игровой мир визуально, все его действия должны реализовываться в рамках трехмерной модели. Так как в этой книге я рассматриваю в основном двухмерные игры, то и представление мира игры у нас будет двухмерным (например, весь мир — это одноэтажный дом). Представьте, что в доме нет крыши и вы смотрите на дом сверху вниз (рис. 14.5). Вы видите комнаты дома, находящиеся в нем объекты и перемещающихся по комнатам людей. Это и есть представление игрового мира, которое требуется для текстовой игры.

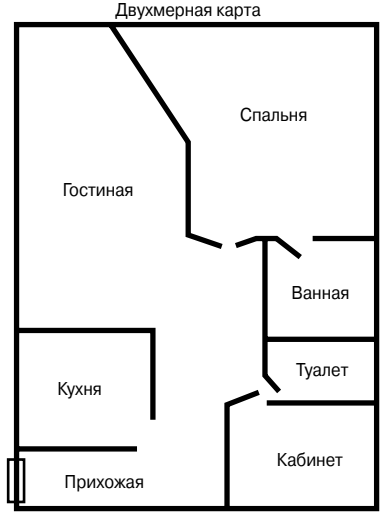


Рис. 14.5. План игрового мира

Реализовать структуру данных такой модели можно двумя способами. Вы можете использовать векторную модель, основанную на линиях и многоугольниках. Она будет представлять собой базу данных геометрии мира игры, и код игры будет использовать ее для передвижения объектов по этому миру. Это очень хороший метод, хотя и несколько сложный для визуализации, поскольку придется работать в чисто векторном пространстве. Так что, пожалуй, для представления мира игры удобнее использовать двухмерную матрицу.

Понятно, что программист никогда не видит создаваемый им мир зрительно, но, имея двухмерный план (в данном случае — расположение комнат на этаже), он сможет легко написать код игры. Например, в игре "Земля теней" для представления мира игры использована двухмерная матрица символов, показанная в листинге 14.8.

Листинг 14.8. Представление мира игры "Земля теней"

```
//этот массив хранит геометрию мира игры

// l - living room (гостиная)
// b - bedroom (спальня)
// k - kitchen (кухня)
// w - washroom (ванная)
// h - hall way (холл)
// r - restroom (комната отдыха)
// e - entry way (прихожая)
// o - office (кабинет)

//      ^
//    NORTH
// < WEST   EAST >
//    SOUTH
//      v

char *universe_geometry[NUM_ROWS]=
{
"********************************",
"*111111111*bbbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbbbbbbbbbbbbbbbb*",
"*1111111111*bbbb*rrr************",
"*111111111hhhhhh*rrrrrrrrrrrrr*",
"*111111111hhhhhh*rrrrrrrrrrrrr*",
"*1111111hhh******rrrrrrrrrrrrr*",
"*******hhhh*rrrrrrrrrrrrrrrrr*",
"*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
"*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
"*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
"*kkkkkkkhhhhh******************",
"*kkkkkkkhhhhhhhhhhhwwwwwwwwwww*",
"*kkkkkkkhhhhhhhhhhhwwwwwwwwwww*",
"*kkkkkkk*hhhhhhhhhhwwwwwwwwwww*",
"*kkkkkkk*hhhh*ooooo*************",
"*kkkkkkk*hhhh*oooooooooooooooo*",
"*kkkkkkk*hhhh*oooooooooooooooo*",
"*kkkkkkk*hhhh*oooooooooooooooo*",
```

```
"********hhhhh*ooooooooooooooooo*",
"*eeeeeeeeeee*ooooooooooooooooo*",
"*eeeeeeeeeee*ooooooooooooooooo*",
"*eeeeeeeeeee*ooooooooooooooooo*",
"*******************************"};
```

Конечно, в таком виде нелегко представить расположение комнат, но если на минутку прищурить глаза, то можно "увидеть" все комнаты. Обратите внимание, что стены указаны символом *, а комнаты — разными символами, например символ o использован для кабинета. Такой способ помогает определить местонахождение игрока. Если в данный момент игрок находится в месте с координатами (x,y), то код, который сообщает, где находится игрок, должен просто проиндексировать массив и определить символ, соответствующий местоположению игрока.

Размещение объектов в мире игры

Размещение объектов в мире игры осуществляется двумя способами. В соответствии с первым необходимо иметь массив или связанный список объектов, где каждый элемент определяет объект, его местоположение и прочие свойства. Единственный недостаток этого метода состоит в том, что, когда игрок находится в комнате и запрашивает ее обзор, требуется обращение ко всему списку объектов, а также определение видимости каждого объекта. Кроме того, обнаружение столкновений, реализация действий "putting" (положить) и "getting" (взять) оказываются более сложными, когда данные об объекте хранятся таким способом. Более простой метод предусматривает наличие другой двухмерной символьной матрицы с той же геометрией, что и у карты мира игры, но вместо стен и полов на этой карте размещают объекты. Затем в коде игры две эти карты, по сути, накладываются одна на другую. В листинге 14.9 представлено размещение объектов на карте игры "Земля теней".

Листинг 14.9. Объекты мира игры "Земля теней"

```
// Этот массив хранит объекты в мире игры

// l - lamp    (лампа)
// s - sandwich (сэндвич)
// k - keys    (ключи)
//       ^
//     NORTH
// < WEST   EAST>
//     SOUTH
//       v

char *universe_objects[NUM_ROWS]=
{"                    ",
 " l                 k ",
 "                     ",
 "                     ",
 "                     ",
 "                     ",
 "                     ",
 "                     ",
 "                     ",
 " l                   ",
 "                     ",
```

```
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
"                ",
" s              ",
"                ",
"                ",
"                ",
"                ",
"     s          ",
"               l",
"                ",
"               "};
```

Из листинга видно, что карта объектов очень разреженная, но так и должно быть, поскольку она содержит только объекты. Зато по ней легко определить близость игрока к объекту, так как существует однозначная взаимосвязь между геометрией мира игры и объектами игры, а также положением игрока. Единственное "но" в использовании такого метода представления миры игры заключается в том, что данный мир имеет конечное число клеток и, следовательно, позиций внутри мира игры, но это такое мизерное неудобство, что на него не стоит обращать внимания.

Выполнение действий

В текстовой игре события выражаются в выводе некоторого текста или изменении некоторых значений в структурах данных, например изменении положения игрока. Текстовая игра — это, по сути, "живая" база данных, к которой обращаются и которую модифицируют в ходе игры. Конечно, помимо текстовых игр, существуют диалоговые игры, но там используется более сложное устройства вывода результата игры на экран, а кроме того, игра работает в реальном времени. В текстовой же игре имеется период ожидания, пока пользователь вводит текстовую строку, подразумевающую какое-либо действие. Во время ожидания игра выполняет функцию ввода строки, и тогда мир игры"засыпает".

Перемещения

Движение в текстовой игре сопровождается изменением положения символа игрока, что выражается в модификации значений ряда переменных. Например, если игрок вводит

move north (двигаться на север),

то компьютер выдаст результат

You take a few steps (вы сделали несколько шагов)

и код программы при этом уменьшает координату Y символа игрока (север находится в отрицательном направлении координаты Y), что и является результатом выполнения запрошенного действия. Конечно, новая позиция игрока должны быть проверена: может

быть, игрок на что-то наступил, или столкнулся со скалой, или упал с обрыва, но фактически движение сопровождается только изменением координат.

Система инвентаризации

Система инвентаризации в текстовой игре представляет собой список, содержащий описание всех объектов, которыми владеет игрок. Этот список может быть массивом, связанным списком или другой структурой данных. Если объекты достаточно сложные, то список может представлять собой список структур. В игре "Земля теней" у игрока очень простой инвентарь, и поэтому он представлен просто массивом символов, используемых для описания имущества. Игрок может иметь сэндвич ("sandwich" — s), набор ключей ("keys" — k) и лампу ("lamp" — l). Наличие у игрока этих объектов представлено соответствующими символами в массиве inventory[]. Если игрок просит показать, чем он в данный момент владеет, необходимо просто пройтись по списку и сформировать строку отчета об имуществе игрока. Но откуда игрок может взять объект? Конечно, он забирает его из мира игры. Например, если игрок находится в комнате и увидел (команда saw) ключи, то он может забрать (get) их себе. При этом символ ключей "k" из мира игры удаляется и вносится в инвентарный список игрока. Конечно, выяснить, что именно находится в комнате, и убедиться, что объект в пределах досягаемости игрока, немного сложнее, но основная идея должна быть понятна.

Реализация обзора, звуков и запахов

Реализация человеческих чувств является самой сложной частью текстовой игры, поскольку игрок не может реально видеть, слышать или обонять. Это означает, что, не говоря уже о трудностях с алгоритмами, описания игры должны быть насыщены прилагательными, наречиями, причастиями — словом, такими частями речи, которые способствуют созданию внутренних образов. Реализация звуков и запахов является самым легким из всей гаммы чувств, поскольку они не сфокусированы. Под фокусированием я понимаю локализацию свойства, т.е. если конкретная комната имеет общий характерный для нее запах, то игрок должен почувствовать его в любом месте комнаты. Аналогичное справедливо и для звука. Если в комнате звучит музыка, то она будет слышна во всех частях комнаты. Сложнее всего программировать обзор, поскольку в сравнении со звуком и запахом он более сфокусирован и, следовательно, труднее поддается реализации. Давайте посмотрим, как можно программировать обзор, звук и запах.

Звук

В текстовой игре мы встречаемся с двумя типами звука: фоновым и динамическим. Фоновые звуки всегда присутствуют в комнате, а динамические могут появляться и исчезать. Вначале поговорим о фоновых звуках. Для их реализации необходимо наличие структуры данных, которая содержит набор строк с описаниями для каждой комнаты. Когда игрок изъявит желание "слышать", компьютер проверит, в какой комнате находится игрок, выберет соответствующие строки и выведет их. Например, если игрок наберет слово "слушать" в машинном цехе, то может состояться следующий диалог:

Что вы хотите сделать? — Слушать
...Вы слышите звуки больших работающих машин. Звуки настолько сильны, что вызывают у вас ощущение зубной боли. Однако где-то в глубине на фоне этого шума, вы слышите странный приглушенный звук, но не можете понять, что это...

Выбор структуры данных, содержащей эти статические строки, зависит от вас. Это может быть массив строк с полем, указывающим описываемое помещение, или массив

структур с отдельными элементами для каждой комнаты. В листинге 14.10 показаны структуры данных, использованные для описания звуков и запахов в игре“Земля теней”.

Листинг 14.10. Структуры данных для описания звуков и запахов в игре “Земля теней”

```
// Структура для хранения строки, используемой в
// качестве описания звука, запаха, обзора и т.п.
typedef struct info_string_typ
   {
     char type;            // Тип информации
     char string[100];  // Описание
   } info_string, *info_string_ptr;
// Описание запахов в разных комнатах

info_string smells[]={

{'l',"You smell the sweet odor of Jasmine with "
   "an undertone of potpourri. "},
{'b',"The sweet smell of perfume dances within your "
   "nostrils...Realities possibly. "},
{'k',"You take a deep breath and your senses are "
   "tantallized with the smell of"},
{'k',"tender breasts of chicken marinating in a garlic "
   "sauce. Also, there is "},
{'k',"a sweet berry smell emanating from the oven."
   "                    "},
{'w',"You are almost overwhelmed by the smell of bathing "
   "fragrance as you"},
{'w',"inhale.                                    "
   "               "},
{'h',"You smell nothing to make note of. "},
{'r',"Your nose is filled with steam and the smell "
   "of baby oil... "},
{'e',"You smell pine possibly from the air coming thru a "
   "small orifice near"},
{'e',"the front door.                            "
   "                  "},
{'o',"You are greeted with the familiar odor of burning "
   "electronics. As you inhale"},
{'o',"a second time, you can almost taste the rustic smell "
   "of aging books.       "},
{'X',""}, // Конец описаний
};

// Описание звуков в разных комнатах

info_string sounds[]={

{'l',"You hear the faint sounds of Enigma playing in "
   "the background along with"},
{'l',"the wind howling against the exterior of the "
   "room.                  "},
```

```
{'b',"You hear the wind rubbing against the window "
    "making a sound similar to"},
{'b',"sheets being pulled off a bed.                "
    "                  "},
{'k',"You hear expansion of the hot ovens along with "
    "the relaxing sounds produced"},
{'k',"by the cooling fans.                          "
    "                    "},
{'w',"You hear nothing but a slight echo in the "
    "heating ducts."},
{'h',"You hear the sounds of music, but you can't make "
    "out any of the words or"},
{'h',"melodies.                                     "
    "                  "},
{'r',"You hear the sound of driping water and the "
    "whispers of a femal voice"},
{'r',"ever so faintly in the background.            "
    "                "},
{'e',"You hear the noises of the outside world muffled by "
    "the closed door to the"},
{'e',"South.                                        "
    "                  "},
{'o',"You hear nothing but the perpetual hum of all the "
    "cooling fans within the"},
{'o',"electronic equipment.                         "
    "                  "},
{'X',""},
};
```

Из листинга видно, что для каждой комнаты имеются свои строчки, а конец массива определяется символом 'X'. Я показал вам только один из способов создания звуков; вы можете использовать свои собственные.

Динамические звуки реализовать сложнее, чем статические, поскольку они могут перемещаться в пределах окружающей среды. В игре “Земля теней” динамические объекты отсутствуют, и поэтому я объясню вам принципы реализации звуков без конкретных примеров. Каждый объект, который может перемещаться в пределах мира игры, имеет характерный лишь для него звук, который определяется строкой, описывающей этот звук. Поэтому, когда игрок просит прослушать звуки в комнате, игра сначала выводит информацию о статических звуках, а затем определяет, какие объекты находятся в комнате. Информация о звуках этих объектов выводится после статической части текста. Конечно, желательно составлять сложные, связанные между собой предложения, которые гораздо лучше воспринимаются игроком, чем отдельные фразы о тех или иных звуках.

Запах

Запахи программируют аналогично звукам, с тем отличием, что строки текста описывают не звук, а запах. Динамические запахи реализуют так же, как и звуки: присоединением запаха к динамическому объекту. Например, людоед должен иметь жуткий запах, описание которого присоединяется к описанию статического запаха в комнате. Для этого сначала определяется наличие людоеда в той же комнате, в которой находится игрок. Это очень просто сделать: достаточно найти на карте мира игры местоположение игрока и людоеда и определить, находятся ли они в ячейках одного и того же типа. Обратите вни-

мание на один чисто художественный аспект описания запаха: если что-то плохо пахнет, то следует приложить максимум усилий для такого описания объекта, которое было бы интересно читать.

Обзор

Реализация обзора в текстовой игре представляет собой очень интересную проблему. Созданный нами виртуальный персонаж должен обладать способностью смотреть в виртуальном мире, в то время как сам игрок не видит ничего, кроме текста. Для программистов нет ничего невозможного — нужно только правильно выбрать алгоритм и структуру данных. Самой простой структурой данных для реализации обзора является карта мира игры, так как игрок передвигается в двухмерной плоскости. Алгоритм также достаточно прост: главное — понять, как мы смотрим и что видим, и затем описать то, что мы видим.

Для начала отметим, что здесь также следует различать статические и динамические объекты. Статический обзор представляет собой общую картину мира. Когда игрок просит осмотреть комнату, то первая часть описания должна касаться статических объектов, и это описание должно быть всегда одинаковым для данных координат игрока. Во второй части описания необходимо сделать упор на объектах, которые находятся в виртуальном поле зрения игрока. Это самая сложная часть алгоритма. Мы должны каким-то образом определить зону видимости, которая располагается непосредственно перед игроком, и выяснить, какие объекты в ней находятся. В отличие от проверки при работе со звуком и запахом мы не можем просто проверить наличие динамического объекта в комнате, так как он может располагаться позади игрока и быть для него невидимым. Поэтому мы должны проверить присутствие объекта в комнате перед игроком.

Для этого необходимо определить пять вещей.

1. Положение игрока.
2. Направление игрока.
3. Позиции динамических объектов.
4. Глубину обзора.
5. Угол зрения.

Первые три задачи легко решаются путем просмотра структур данных для игрока и динамических объектов. Вопросы возникают только с пунктами 4 и 5. Глубина обзора определяет, насколько далеко видит игрок. Эта величина весьма относительна, но общее правило гласит, что игрок может видеть по меньшей мере в пределах всей длины самой большой комнаты. Немаловажным фактором является угол зрения, т.е. под каким углом находятся видимые для игрока объекты (рис. 14.6).

Поскольку игрок может быть обращен только к четырем частям света (север, юг, запад, восток), сканирование видимых объектов упрощается. Для реализации обзора нам требуются всего лишь два цикла for. Область сканирования будет иметь форму перевернутой пирамиды, а угол зрения будет равен 90°, что довольно близко к обычному человеческому полю зрения. Таким образом, сканирование начинается с позиции игрока и проверяет каждую ячейку в области обзора (рис. 14.7).

В общем случае один цикл for управляет сканированием вдоль оси X, а второй — вдоль оси Y. Сканирование будет продолжаться до тех пор, пока не будет достигнута заданная глубина сканирования. При сканировании области обзора проводится проверка на наличие в ней динамических объектов. После того как программа выводит информацию о статическом обзоре помещения, она обращается к полученному в результате сканирования списку и выводит строки с информацией о видимых динамических объектах.

Такой метод использован, например, в игре “Земля теней” для обзора объектов в комнате. Ниже приведен листинг 14.11, в котором представлен код, сканирующий область обзора в северном направлении. Обратите внимание на структуру циклов for и выполняемые проверки. Не беспокойтесь об отсутствующих определениях переменных — просто постарайтесь понять действие фрагмента кода в целом.

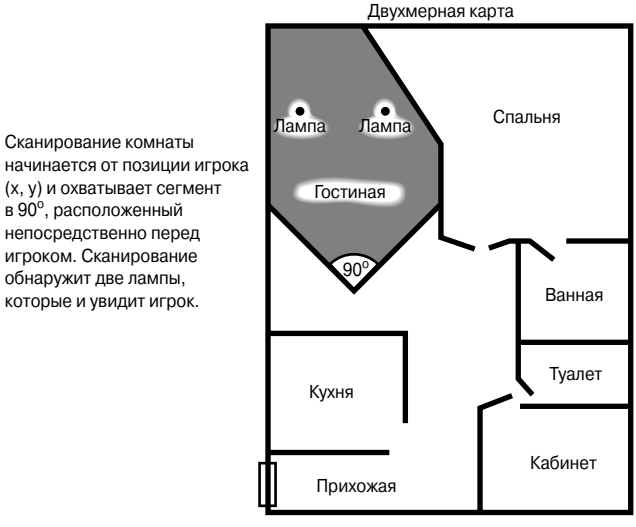


Рис. 14.6. Виртуальный обзор в действии

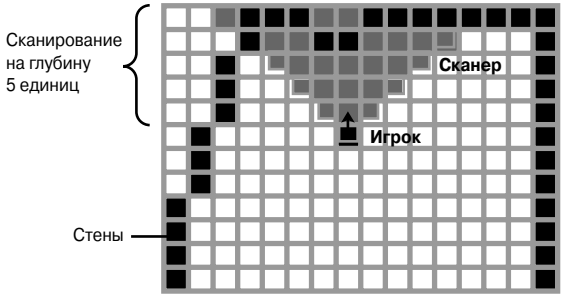


Рис. 14.7. Сканирование области обзора игрока

Листинг 14.11. Фрагмент кода сканирования поля обзора в северном направлении

```
case NORTH:
{
// сканируем следующим образом:
// .....
//  ...
//   P
   for(y = you.y, scan_level = 0; y >= (you.y-depth);
      y--, scan_level++)
```

```
  {
    for(x = you.x - scan_level;
      x <= you.x + scan_level; x++)
    {
      // (x,y) - проверяемая точка. Убедимся, что она
      // находится в границах игровой вселенной и в
      // пределах одной и той же комнаты
      if (x >= 1 && x < NUM_COLUMNS-1 &&
        y >= 1 && x < NUM_ROWS-1 &&
        universe_geometry[y][x] ==
        universe_geometry[you.y][you.x])
      {
        // Проверим наличие объекта в точке
        if (universe_objects[y][x]!=' ')
        {
          // Вставим объект в список
          stuff[*num_objects].thing =
            universe_objects[y][x];
          stuff[*num_objects].x    = x;
          stuff[*num_objects].y    = y;
          // Увеличиваем количество объектов
          (*num_objects)++;
        } // if
      } // if
    } // for x
  } // for y
  // Возвращаем количество найденных объектов
  return(*num_objects);
} break;
```

Функция сканирует область перед игроком и проверяет, находится ли каждая просматриваемая ячейка в пределах мира игры. При положительном результате функция проверяет, нет ли в данной ячейке какого-либо объекта. Если такой объект есть, то он вносится в список видимых объектов. Обратите внимание на код, который определяет, находится ли сканируемый блок в пределах мира игры. Выполнение такой проверки абсолютно необходимо, поскольку сканирование осуществляется до определенного расстояния, и проверяемые ячейки могут оказаться за пределами игрового пространства. И не забывайте, что вывод информации о динамических объектах после вывода информации о статических объектах должен представлять собой связный рассказ.

Игра в реальном времени

До сих пор мы рассматривали технологию текстовой игры не в реальном масштабе времени. Это обусловлено тем, что игра ждет, пока пользователь не введет очередную текстовую строку, затем эта строка обрабатывается, логика игры выполняет ее, и цикл начинается сначала. Проблема в том, что во время фазы ввода текста игра находится в состоянии ожидания. В результате время "останавливается" до тех пор, пока игрок не введет строку и не нажмет клавишу <Enter>. В это время динамические объекты в игре не могут перемещаться. Иногда это желательно, иногда — нет. Существуют текстовые игры, которые останавливаются и ждут ввода игрока; другие же работают во время ожидания ввода текста.

Сделать игру работающей в реальном времени очень легко: нужно всего лишь слегка изменить функцию ввода, чтобы она напоминала обработчик клавиатуры, т.е. посылала строку на вход синтаксического анализатора лишь при нажатии клавиши <Enter>. Цикл событий игры в этом случае может выглядеть следующим образом:

```
while(!done)
{
  Get_Next_Input_Character(); // Ввод очередного символа
  if (Пользователь ввел строку полностью)
  {
    Parser_Logic();       // Разбор введенной строки
  }

  // Перемещение объектов в игре
  Move_Objects();
}
```

Реальное время в этой ситуации обеспечивается отсутствием ожидания ввода команды пользователя. Если команда введена, она обрабатывается; в противном случае игра продолжается в реальном времени.

Обработка ошибок

Текстовая игра должна обрабатывать ошибки так же, как и диалоговые игры. Однако большинство ошибок обрабатываются при синтаксическом анализе текста. Существуют лексические, синтаксические и семантические проверки текста, и все они должны быть выполнены. Лексические тесты просты, поскольку они осуществляются сравнением введенных строк и слов из словаря. Синтаксические и семантические тесты сложнее, поскольку логика игры должна использовать для выяснения корректности введенного предложения продукции и правила языка. Это сложный процесс, требующий к тому же отдельного рассмотрения множества частных случаев и исключений (та же ситуация возникает и при разработке компиляторов или интерпретаторов, где обрабатываемый исходный текст программы очень часто содержит множество ошибок, которые обязательно нужно обнаружить). В текстовой игре лучше перестраховаться, чем пропустить ошибку и потом сожалеть об этом. Поэтому, чем больше фильтров, ловушек и тестов имеет входной синтаксический анализатор, тем надежнее будет работать игра.

Игра “Земля теней”

Теперь, когда вы можете считать себя специалистом в написании текстовых игр, мне бы хотелось показать вам конкретный работоспособный пример такой игры. Это полнофункциональная текстовая игра “Земля теней”, которая позволяет вам взаимодействовать с окружающей средой. Идея игры проста: вы должны найти ключи и войти в кабинет. Среда, в которой вы должны играть, — моя старая квартира в Силиконовой Долине. Я смоделировал свою квартиру с ее обычным видом, звуками и запахами.

По условию игры вы являетесь невидимой тенью, которая может свободно и незаметно для окружающих передвигаться в квартире. Словарь и грамматика игры очень просты, как и действия, выполняемые вами. Если вы разберетесь с игрой “Земля теней”, то без проблем сможете создать нечто похожее самостоятельно — с соответствующим планированием, структурами данных и алгоритмами. Кроме того, если вы хотите заняться ролевыми играми, то просто обязаны разобраться в том, как работает эта игра.

Язык "Земли теней"

Словарь игры "Земля теней" уже был приведен в табл. 14.1, но сейчас речь пойдет не о нем, а о синтаксисе языка. Попробуем изложить правила языка в упрощенном виде. Начнем с перечисления всех слов словаря[4].

- Объекты игры: LAMP (лампа), SANDWICH (сэндвич), KEYS (ключи).
- Направления: EAST (восток), WEST (запад), NORTH (север), SOUTH (юг).
- Относительные направления: FORWARD (вперед), BACKWARD (назад), RIGHT (направо), LEFT (налево).
- "Действия" (глаголы языка) игры: MOVE (двигаться), SMELL (нюхать), LOOK (смотреть), LISTEN (слушать), PUT (положить), GET (взять), EAT (есть), INVENTORY (список имущества), WHERE (где?), EXIT (выход).
- Артикли или соединительные слова языка: THE.
- Предлоги: IN (в), ON (на), TO (к), DOWN (вниз).

Начнем с разрешенных форм глаголов, т.е. посмотрим, каким должно быть употребление глаголов в соответствии с правилами английского языка. Некоторым глаголам не нужны объекты. В нашем языке к таким глаголам относятся: smell, listen, inventory, exit и where. Если написать любой из этих глаголов сам по себе, это будет корректной командой. Кроме того, если после глаголов стоят предложные группы, артикли или дополнения, то они являются своего рода предупредительным сигналом, т.е. указывают, что за глаголом действия последует объект. Результат использования этих глаголов в качестве команд приведен ниже.

smell	Описание запахов в комнате, в которой вы находитесь
listen	Описание звуков в комнате, в которой вы находитесь
inventory	Вывод списка вашего имущества
where	Описание вашего местоположения в доме, а также направление вашего движения
exit	Завершение игры

Следующий набор глаголов действия используется вместе с объектами, прилагательными или предложными группами. Это глаголы move, put, get, look, eat. Ниже приведены разрешенные формы использования данных глаголов в виде синтаксических продукций (скобки означают, что слово(а) является необязательным, а "|" означает логическое ИЛИ).

move + (относительное направление) | move + to + относительное направление | move + to the + относительное направление.

В соответствии с этой продукцией следующее предложение совершенно корректно:

move to the right (двигаться направо).

В результате игрок переместится на шаг вправо. Другой возможный вариант предложения выглядит так:

move (двигаться).

В результате игрок будет двигаться в том направлении, куда он обращен лицом. Примером же некорректного предложения может служить следующее:

move to the east (двигаться на восток).

[4] Заметим, что отнесение слов к той или иной части речи соответствует грамматике английского языка. — *Прим. ред.*

Это предложение некорректно, поскольку east (восток) — это часть света, а не относительное направление движения.

Следующий интересный глагол действия — это look (смотреть), продукция для предложений с которым выглядит следующим образом:

look + (направление) | look + to + направление | look + to the + относительное направление.

Используя этот глагол, игрок может видеть объекты в комнате. Если игрок использует слово look без указания направления, то он получит только статическое описание помещения. Чтобы увидеть объекты, игрок ***должен*** указать направление. Например, чтобы увидеть северную часть комнаты, игроку необходимо ввести **look to the north**, или **look north**, или **look to north**. Все перечисленные формы предложения эквивалентны. В результате будет выведена информация об объектах, попавших в поле зрения игрока.

Игрок начнет игру, стоя лицом на север, поэтому необходима команда для поворота игрока. Поворот игрока реализуется с помощью глагола действия turn (повернуться), предложение с которым строится аналогично предложению с глаголомlook.

turn + (направление) | turn + to + направление | turn + to the + относительное направление.

Следовательно, чтобы повернуться на восток, игрок должен ввести команду **turn to east**.

Следующие два глагола действия — put (положить) и get (взять) — связаны с манипуляциями объектами. Их обычно используют, чтобы положить или поднять предметы. Единственными корректными объектами в данной игре являются ключи, лампа и сэндвич. Ниже приведены правила построения предложения для каждого из этих глаголов действия.

put + объект | put + down + объект | put + down the + объект
get + объект | get + the + объект

Глаголы put и get используют для перемещения объектов в пространстве. Представим, например, что игрок увидел перед собой связку ключей. Он может ввести команду **get the keys** (взять ключи) и поднять ключи. Если игрок захочет положить ключи, он может сделать это с помощью фразы **put the keys** (положите ключи).

И наконец, еще один глагол действия — eat (есть). Правило для eat следующее:

eat + объект | eat + the + объект

Так, если у вас появится желание съесть лампу, просто напишите **eat the lamp**, и потребность организма в железе будет удовлетворена на много дней вперед!

Поначалу грамматика и ограниченный словарь могут показаться вам слишком скучными, но через некоторое время все встанет на свои места и будет казаться совершенно естественным. Конечно, при желании вы можете добавлять новые слова в словарь и вводить новые грамматические правила. И последний штрих: синтаксический анализатор не чувствителен к регистру, поэтому можно использовать символы как верхнего, так и нижнего регистров.

Построение “Земли теней” и игра в нее

“Земля теней” — единственная полностью переносимая игра в книге. Это обусловлено тем, что она является чисто текстовой. Впрочем, сделаю две оговорки: в “Земле теней” используется функция kbhit() и цветной текстовый драйвер ANSI. Этот драйвер применяли давно, еще во времена DOS, чтобы иметь возможность использовать специальные управляющие последовательности для изменения цветов в чисто текстовых приложениях. Для его подключения в файл config.sys требуется добавить строку

```
DEVICE=C:\WINDOWS\COMMAND\ANSI.SYS
```

Если этот драйвер находится в другом месте, укажите в данной строке его полное имя. Если не считать этого, код игры представляет собой код на чистом C/C++, без использования графики или машинно-зависимых вызовов. Для создания программы просто скомпилируйте исходный файл SHADOW.CPP в консольное приложение и скомпонуйте его со стандартными библиотеками C/C++; больше от вас ничего не требуется. Впрочем, на прилагаемом компакт-диске находится уже скомпилированный файлSHADOW.EXE.

Теперь вы знаете практически все, что нужно для начала игры. Однако позволю себе дать несколько советов. Игра начинается с вопроса о вашем имени, затем следует вопрос о ваших действиях. В этот момент вы находитесь у входа в квартиру. Налево от вас расположена кухня, а перед вами (на север от вас) — прихожая. Попробуйте перемещаться в пространстве с помощью команды **move**, убедитесь, что вы слышите звуки и чувствуете запахи. Не забывайте, что команда **move** всегда перемещает вас в том направлении, к которому вы обращены лицом, а команде **look** необходимо указать, в каком направлении вы желаете смотреть.

Цикл игры “Земля теней”

Цикл игры очень прост, поскольку действие происходит не в реальном времени. Он приведен ниже в листинге 14.12.

Листинг 14.12. Цикл игры “Земля теней”

```
void main(void)
{

// Вызываем заставку игры
Introduction();

printf("\n\nWelcome to the world of SHADOW LAND...\n\n\n");

// Получаем имя пользователя
printf("\nWhat is your first name?")
scanf("%s", you.name);
// Главный цикл событий.
// Предупреждение: он выполняется НЕ в реальном времени
while(!global_exit)
{
  // Запрос действий пользователя
  printf("\n\nWhat do you want to do?");

  // Получаем строку текста
  Get_Line(global_input);

  printf("\n");
  // Разбираем введенное предложение
  Extract_Tokens(global_input);

  // Анализируем глагол и выполняем команду
  Verb_Parser();

} // while

printf("\n\nExiting the universe of SHADOW LAND... "
```

```
 "see you later %s.\n", you.name);

//восстанавливаем цвет экрана
printf("%c%c37;40m", 27, 91);

} // main
```

Функция main() начинает работу с вывода экрана заставки, а затем запрашивает у игрока его имя. После этого игра входит в главный цикл событий. Цикл ожидает ввода пользователя, который получает с помощью вызова функции Get_Line().

Затем строка преобразуется в токены с помощью функции Extract_Tokens(), после чего выполняется синтаксический анализ посредством вызова функции Verb_Parser(). Этот цикл повторяется всякий раз, когда игрок вводит строку текста, до тех пор, пока введенная строка не будет содержать команду **exit**.

Победитель игры

Я думаю, что уже достаточно рассказал вам об игре и теперь вы сами догадаетесь, как стать в ней победителем. Удачи вам!

Вы можете дополнить игру новыми возможностями. Введите в нее новые объекты, новые слова. Если у вас все будет получаться, попробуйте создать графическое представление комнаты, игрока, объектов — и вы получите полноценную графическую игру!

Резюме

В этой главе вы освоили азы создания компиляторов и научились применять эти знания для разработки текстовых игр. Вы познакомились с тем, как представлять мир текстовой игры, как реализовать обзор, звуки, запах и как сделать текстовую игру увлекательной. И наконец, вы получили возможность посмотреть, как я живу!

ГЛАВА 15

Разработка игры Outpost

Это последняя глава книги, и я думаю, для нее наступило самое время. В этой главе в общих чертах описана разработка и реализация несложной игры — *Outpost*, которую я написал с использованием технологий, рассмотренных в этой книге.

Написание этой игры занимает около пяти дней. Несмотря на такой короткий срок, она использует трехмерные спрайты, системы частиц, звуковые эффекты, несколько разных типов противника... Игра действительно достаточно проста для того, чтобы можно было в ней что-то изменить, если вы этого захотите. В данной главе рассматриваются следующие вопросы.

- Начальный эскиз игры *Outpost*
- Инструменты, используемые при написании игры
- Мир игры и его прокрутка
- Космический корабль игрока
- Поле астероидов
- Противники
- Запасы энергии
- Системы частиц
- Работа игры
- Компиляция игры *Outpost*

Начальный эскиз игры

Я хотел создать игру, которую было бы легко написать, которая хорошо бы смотрелась, имела достаточно примитивный сценарий и использовала прокрутку пространства. Поэтому я выбрал игру типа *Asteroids* — хотя бы потому, что у нее очень простой черный фон. Кроме того, искусственный интеллект астероидов, равно как и искусственный интеллект противников типа “найти и уничтожить” я сделал достаточно простым, что соответствует поставленной цели — создать несложную игру.

Сюжет игры

Фабула игры такова: вы пилот сверхсекретного космического корабля “Wraith”, оснащенного супермощными видами оружия. Вас отправили в сектор Alpha 11, и вашей задачей является освобождение этого сектора от вторжения враждебных пришельцев. Пришельцы окружили весь сектор аванпостами, которые вы должны уничтожить. Проблема в том, что сектор заполнен крупными летающими каменными обломками (астероидами), изобилует боевыми кораблями пришельцев, а каждый аванпост защищен самонаводящимися минами. Таков сюжет игры (как в настоящем кинофильме, не так ли?). На рис. 15.1 и 15.2 показаны заставка игры и одно из изображений, появляющихся в процессе игры. Хотя я и считаю, что в большинстве случаев сюжет игры является наименее важным этапом написания программы, тем не менее утверждаю, что именно сюжет — это тот стержень, к которому “привязывается” написание самой игры.

Рис. 15.1. Заставка игры Outpost

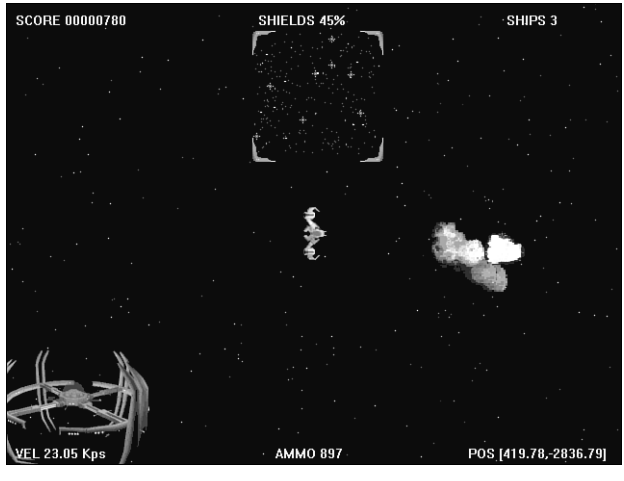


Рис. 15.2. Outpost в процессе игры

Проектирование хода игры

Поскольку сюжет игры уже создан, приступим к проектированию хода игры. Это проектирование заключается в разработке "правил поведения" объектов игры: какие функции выполняют те или иные объекты или элементы управления, каковы их задачи и т.д. Игра не играется сама по себе, и я должен продумывать наперед, что может сделать игрок или его противники в процессе игры, решать, по каким признакам игрок считается выигравшим (или проигравшим), каким видом искусственного интеллекта должен быть наделен каждый игровой объект, и многое другое. Поскольку игра *Outpost* не имеет разных уровней или нескольких разных видов стратегии игры, мы не будем рассматривать методы проектирования этих особенностей.

Инструменты, используемые при написании игры

Как и во всей этой книге, при написании игровой программы я использовал 256-цветный режим, чтобы упростить изложение. Я воспользовался Caligari TrueSpace IV (TS4), которая, по моему мнению, является для своей цены лучшей в мире программой для разработки трехмерной графики. Демонстрационная копия TS4 находится на прилагаемом компакт-диске.

Двухмерная графика создавалась и отлаживалась с использованием отличного графического пакета JASC Paint Shop Pro 5.1, который поддерживает различные дополнительные возможности, предоставляемые в виде отдельных модулей, и имеет самый простой интерфейс, который я когда-либо видел. Демонстрационная копия этого пакета также находится на прилагаемом компакт-диске.

И наконец, звуковые эффекты заимствованы из разных источников и обработаны с помощью пакета Sound Forge XP компании Sonic Foundry, который является одним из лучших пакетов, предназначенных для редактирования звуковых эффектов на персональных компьютерах. Демонстрационную копию пакета Sound Forge также можно найти на прилагаемом компакт-диске. Кроме упомянутых выше инструментов, я использовал также VC++ 5.0, 6.0 и DirectX.

В табл. 15.1 перечислены все игровые объекты и методы создания каждого из них.

Таблица 15.1. Объекты игры Outpost

Объект	*Метод создания*
Астероиды	С помощью TS4
Космический корабль игрока	Рисование вручную с помощью PSP
Аванпосты	С помощью TS4
Мины	С помощью TS4
Боевые корабли	С помощью TS4
Запасы энергии	С помощью TS4
Плазменные импульсы	Рисование вручную с помощью PSP
Звездный фон	Отдельные пиксели
Взрывы	Оцифрованы с помощью Pyrotechnics Stock Media

Сетки всех трехмерных объектов создавались вручную, и при использовании TS4 работа над каждой из них занимала менее часа. Создание каждого изображения, которое рисовалось вручную, также занимало около часа. Вращение космического корабля игрока я предпочел осуществить с использованием программного алгоритма вращения PSP, а не вручную. Все звуковые эффекты были приведены к 8-битовому монозвуку с частотой оцифровки 11 kHz.

Мир игры: прокрутка пространства

Я уже рассказывал о том, каким образом осуществляется прокрутка пространства, так что здесь я не буду повторяться. Тем не менее в игре *Outpost* есть пара интересных особенностей, касающихся прокрутки. Прежде всего, игрок всегда находится в центре экрана. Данное обстоятельство несколько упрощает игру, но главное — дает игроку возможность перемещаться во всех направлениях, способствуя максимально правильной реакции на ситуации, возникающие по ходу игры. Если игрок может добраться до края экрана (в случае игры с перемещением игрока по пространству), то понятно, что возможна ситуация, когда неожиданно возникший противник успеет поразить игрока, даже если реакция последнего будет всего лишь несколько миллисекунд. Расположение игрока в центре экрана дает ему хороший обзор пространства и всего того, что делается вокруг него.

Что касается размера игрового мира, то я выбрал его размер равным 16000×16000, как видно из представленного ниже фрагмента кода.

```
// Размер игрового пространства
#define UNIVERSE_MIN_X  (-8000)
#define UNIVERSE_MAX_X   (8000)
#define UNIVERSE_MIN_Y  (-8000)
#define UNIVERSE_MAX_Y   (8000)
```

Выбор того или иного размера пространства игры всегда сложен. Я использую стандартную технологию: сначала оцениваю частоту вывода кадров, затем максимальную скорость перемещения игрока, после чего, основываясь на этих данных, принимаю решение о том, сколько времени необходимо дать игроку, чтобы он мог добраться из одного края пространства в другой.

Следовательно, если игрок перемещается с максимальной скоростью 32 пикселя за кадр, частота кадров — 30 fps, а время перемещения игрока из одного конца пространства в другой составляет 10 секунд, вы получите размер игрового пространства, равный 32*30*10 = 9600 пикселей.

Естественно, значения размеров нашего игрового мира немного другие, но я просто хотел показать вам принцип расчета размеров на основании условий игры.

Другая проблема, требующая обсуждения при написании игры, в которой используется прокрутка пространства, — это проблема заселенности. Вам может показаться, что пространство размером 10×10 секунд лета, или 16000×16000 пикселей не такое уж большое, но, чтобы заполнить его, вам потребуются сотни, если не тысячи астероидов — иначе игроку придется долго летать, чтобы найти хоть один из них.

Что касается алгоритма прокрутки, то тут нечего много говорить. Позиция игрока используется в качестве центра окна вывода, как показано на рис. 15.3.

Алгоритм получает позицию игрока, после чего переносит его в центр экрана вместе с близлежащими объектами. Объекты, которые расположены в пределах окна, визуализируются, все остальные — нет. Таким способом не отображается только один вид объектов — звезды, по сути, являющиеся отдельными пикселями, которые просто переносятся на противоположный край экрана при выходе за его границы. Это, кстати, значит, что

если вы будете медленно передвигаться по оси X или Y и внимательно всмотритесь, то увидите одни и те же звезды в одних и тех же местах.

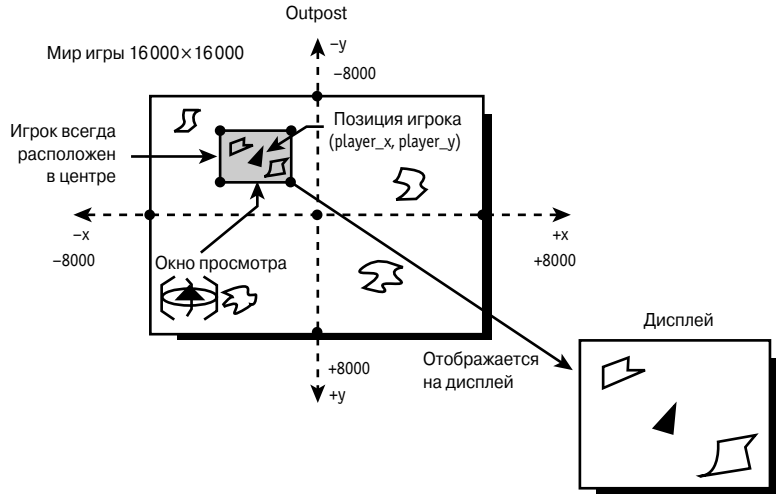


Рис. 15.3. Алгоритм прокручиваемого окна для игры Outpost

Космический корабль игрока

Космический корабль игрока рисуется вручную. Сначала я собирался изобразить его со всеми подробностями, но он имеет так много деталей, что я решил нарисовать только эскиз, а затем воспользовался программой Paint Shop Pro для задания эффектов освещенности, чтобы корабль выглядел более реально. Более того, я нарисовал корабль "Wraith" только в одном ракурсе (направленным на север), после чего использовал встроенную в программу PSP функцию вращения для получения изображений корабля, повернутых на 16 разных углов (рис. 15.4).

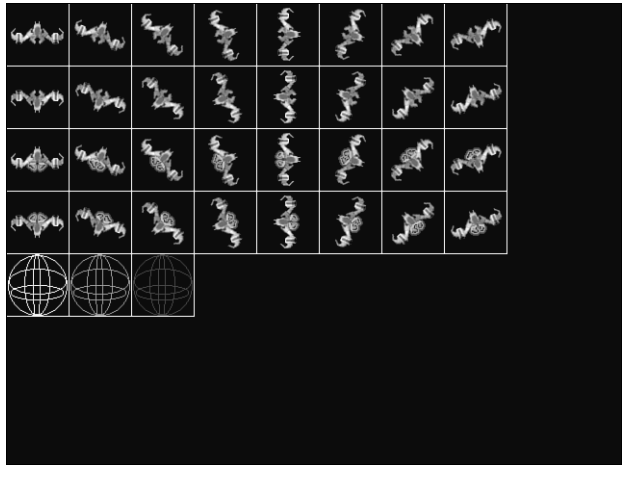


Рис. 15.4. Набор изображений космического корабля

Корабль не совершает никаких операций, кроме перемещений по пространству и стрельбы, однако алгоритм трения представляет определенный интерес. Я хотел, чтобы перемещения корабля "Wraith" выглядели так, как будто он подчиняется воздействию сил инерции, поэтому мне понадобилось ввести трение в процесс поворота корабля. Эту задачу я осуществил с помощью стандартных методик, с которыми вы уже знакомились в предыдущих главах.

Вектор скорости корабля должен изменяться, когда игрок наносит удар по противнику (т.е. стреляет), и это изменение (отдача при стрельбе) зависит от конкретного направления стрельбы. При этом, составляя программу, вы не должны заботиться о конкретной величине и направлении этого изменения, т.е. думать о таких вещах, как, например, "Если корабль летит с такой-то скоростью на восток, а игрок стреляет на север, то как это должно отразиться на перемещении корабля, чтобы все выглядело естественно?". Вместо вас эту задачу математически решает соответствующий программный код.

Ниже представлен фрагмент этого управляющего кода для корабля "Wraith".

```
// Проверка движения игрока
if (keyboard_state[DIK_RIGHT])
{
  // Вращение вправо
  if (++wraith.varsI[WRAITH_INDEX_DIR] > 15)
    wraith.varsI[WRAITH_INDEX_DIR] = 0;

} // if
else
  if (keyboard_state[DIK_LEFT])
  {
    // Вращение влево
    if (--wraith.varsI[WRAITH_INDEX_DIR] < 0)
      wraith.varsI[WRAITH_INDEX_DIR] = 15;

  } // if

// Движение вверх
if (keyboard_state[DIK_UP])
{
  // Движение вперед
  xv = cos_look16[wraith.varsI[WRAITH_INDEX_DIR]];
  yv = sin_look16[wraith.varsI[WRAITH_INDEX_DIR]];

  // Проверка включенности двигателей
  if (!engines_on)
    DSound_Play(engines_id,DSBPLAY_LOOPING);

  // Включить двигатели
  engines_on = 1;

  Start_Particle(PARTICLE_TYPE_FADE,
          PARTICLE_COLOR_GREEN, 3,
          player_x+RAND_RANGE(-2,2),
          player_y+RAND_RANGE(-2,2),
          (-int(player_xv)>>3),
          (-int(player_yv)>>3));

} // if
else
```

```
   if (engines_on)
   {
      // Выключение двигателей и звука
      engines_on = 0;

      // Отключение звука

      DSound_Stop_Sound(engines_id);
   } // if


// Переключение режима отображения
if (keyboard_state[DIK_H] && !huds_debounce)
{
   huds_debounce = 1;
   huds_on = (huds_on) ? 0 : 1;

   DSound_Play(beep1_id);

} // if

if (!keyboard_state[DIK_H])
   huds_debounce = 0;

// Переключение режима сканера
if (keyboard_state[DIK_S] && !scanner_debounce)
{
   scanner_debounce = 1;
   scanner_on = (scanner_on) ? 0 : 1;

   DSound_Play(beep1_id);

} // if

if (!keyboard_state[DIK_S])
   scanner_debounce = 0;

// Изменение скорости игрока
player_xv+=xv;
player_yv+=yv;

// Проверка на максимальную скорость
vel = Fast_Distance_2D(player_xv, player_yv);

if (vel >= MAX_PLAYER_SPEED)
{
   // Пересчет скорости
   player_xv = (MAX_PLAYER_SPEED-1)*player_xv/vel;
   player_yv = (MAX_PLAYER_SPEED-1)*player_yv/vel;
} // if

// Добавление трения

// Перемещение игрока
player_x+=player_xv;
player_y+=player_yv;
```

НА ЗАМЕТКУ

Несколько слов о названии "Wraith" ("Призрак"). Много лет назад я смотрел кинофильм под названием "*The Wraith*", в котором играл Чарли Шин (Charlie Sheen). В фильме Чарли закончил жизненный путь, упав в автомобиле с отвесной скалы. Он умер, но вернулся в качестве злого духа, который мстил своим убийцам. Отсюда я и взял название для космического корабля.

Изучая код, обратите внимание на фрагмент, в котором задано ограничение скорости корабля "Wraith", если он перемещается слишком быстро. По сути, я постоянно контролирую длину вектора скорости, сравнивая ее с максимальной длиной. Если вектор слишком длинный, я уменьшаю его.

Наконец, корабль "Wraith" имеет защитный экран и газовые струи, испускаемые работающими двигателями. Защитный экран — это не более чем битовое изображение, которое я налагаю на Wraith, когда что-либо попадает в него, а факелы представляют собой просто частицы, которые я выпускаю в произвольном порядке при включенных двигателях малой тяги. Эффект работающего двигателя достигается использованием двух изображений для каждого направления движения корабля "Wraith": одно — при включенных двигателях, другое — при выключенных. Когда двигатели включены, я чередую изображения для включенных и выключенных двигателей, и это выглядит, как работа импульсного двигателя.

Поле астероидов

Поле астероидов состоит из большого числа произвольно перемещающихся астероидов трех разных размеров: малого, среднего и большого, которые создавались с использованием программы TS4. На рис. 15.5 показаны астероиды в окне моделирующей программы, а на рис. 15.6 представлен их окончательный вид. Получив объекты, действительно похожие на астероиды, причем должным образом освещенные, я добавил вращение, а затем преобразовал анимационные `.TGA`-файлы в битовые изображения (`.bmp`) и импортировал их в игру.

Физическая модель астероидов довольно проста. Они перемещаются в определенном направлении с постоянной скоростью, до тех пор пока не будут поражены игроком. Кроме того, астероиды вращаются с разными скоростями в зависимости от своей массы. При поражении астероида степень его уничтожения зависит от его размера и прочности. С помощью импульсного взрывателя можно полностью уничтожить даже большой и очень прочный астероид, но иногда он только распадается на осколки, образуя при этом астероиды меньшего размера. Последний вариант определяется двумя факторами: вероятностью и возможностью добавления новых астероидов в поле (астероидов среднего и малого размера может быть только определенное количество, так что вы не всегда сможете развалить большие астероиды на меньшие).

К тому же мне не нравится слишком однозначный способ образования астероидов среднего и малого размера: например, большой астероид всегда распадается на два средних, а средний — обязательно на два малых. Поэтому иногда большой астероид может развалиться на один средний и два малых или два средних и два малых и т.д. Я добавил несколько таких вариантов с вероятностями их осуществления, чтобы сделать игру более интересной. Если вы захотите иметь в игре больше астероидов, необходимо изменить следующую директиву `#define`:

```
#define MAX_ROCKS        300
```

Можете поэкспериментировать, увеличивая число астероидов до 1000 или даже до 10000, если позволит мощность вашего компьютера.

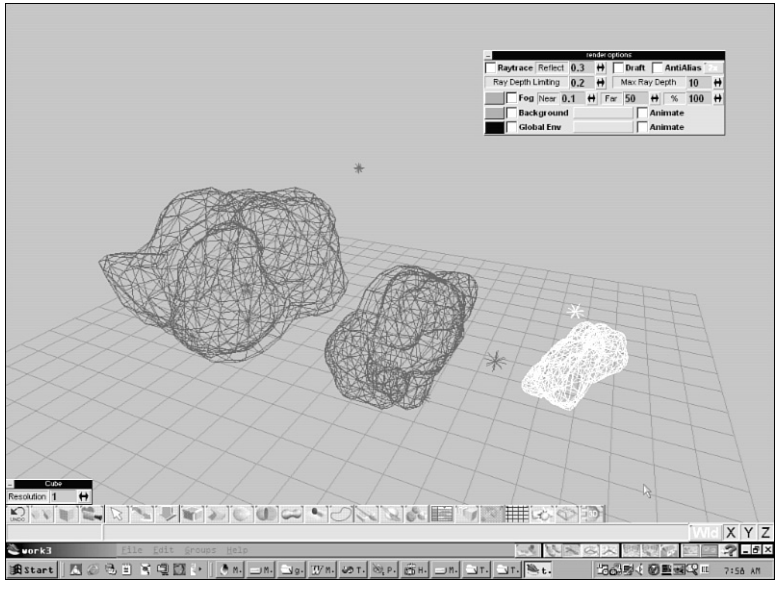


Рис. 15.5. Трехмерные модели астероидов

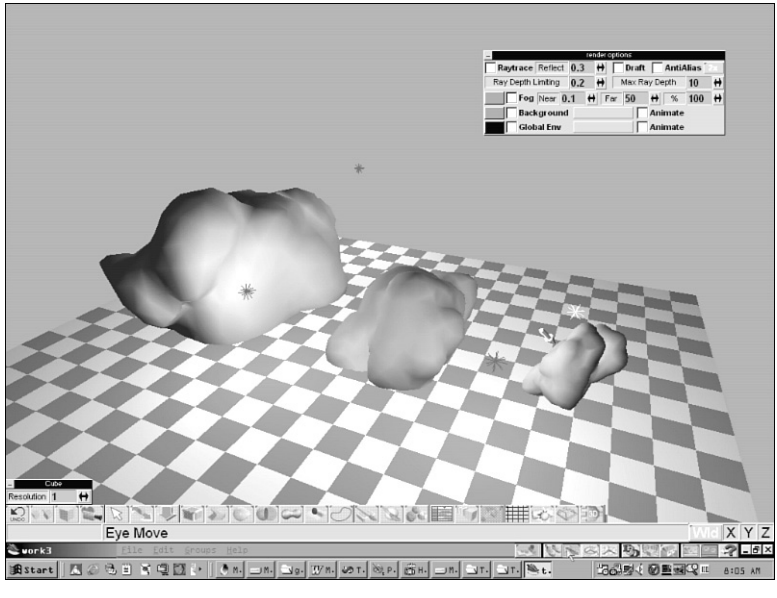


Рис. 15.6. Окончательный вид астероидов

СОВЕТ

Мои симпатии к астероидам выросли из любви к игре *Asteroids*. Мне кажется, эта любовь возникла тогда, когда в колледже я выиграл на пари 100 долларов. Кое-кто из студентов-компьютерщиков утверждал, что я не смогу написать игру *Asteroids* на языке Pascal для компьютера IBM XT быстрее, чем они. Они видели другие игры, которые я написал, но считали, что я их просто откуда-то “передрал”. Конечно, все они были типичными студентами-компьютерщиками, которые не умели ничего, за исключением того, что позволял API.

Я сел и написал *Asteroids* в двухмерном варианте менее чем за восемь часов. Это была точная копия векторной версии Atari (правда, без звуковых эффектов). Я выиграл 100 долларов и не замедлил очистить их карманы.
С тех пор прошло немало времени, но я до сих пор помню код игры *Asteroids* и люблю всегда приводить его в качестве примера. Это похоже на мое “Hello World” в программировании игр. :)

И в заключение нужно сказать, что, когда астероид достигает края игрового пространства, он автоматически переносится на противоположный край. Вы же при желании можете осуществить его отражение от края игрового поля.

Противники

Противники в этой игре не являются каким-то обитающим в пространстве пугалом, наделенным интеллектом, тем не менее они играют свою роль. Большей частью они используют простейшие методы искусственного интеллекта, такие, как детерминированная логика и методы конечных автоматов. Однако здесь также имеется пара новых технологий, которые я использовал для алгоритмов слежения. Вы познакомитесь с ними позже в этой главе, а сейчас давайте посмотрим, каким образом создается и реализуется в игре каждый противник.

Аванпосты

Модель, которую я использовал для создания аванпостов, вероятно, наиболее сложная трехмерная модель во всей игре. Работа над аванпостами заняла у меня несколько часов. Украшением этой модели является масса деталей, видных на рис. 15.7 и 15.8, однако многие из них были утрачены в процессе окончательного вывода изображения модели.

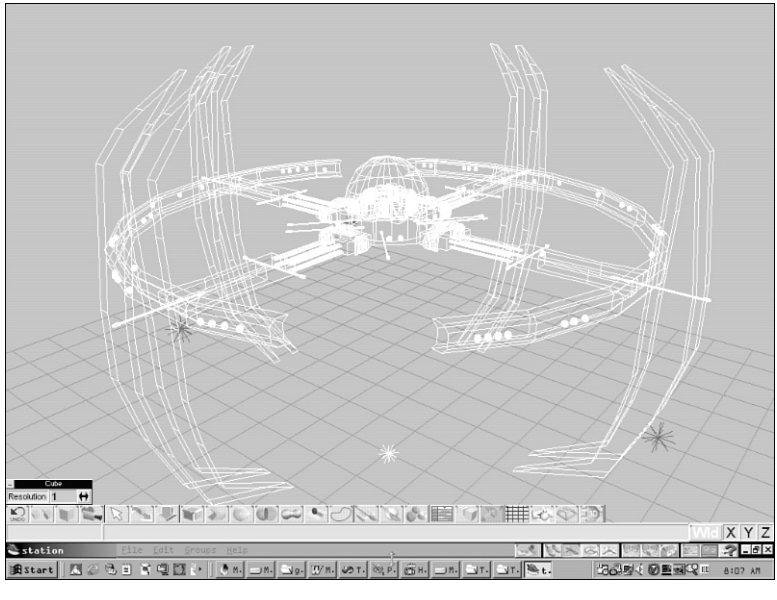


Рис. 15.7. Трехмерная модель аванпоста

Задача аванпостов одна — находиться на своем месте и вращаться. Они не имеют вооружения, искусственного интеллекта, вообще ничего не имеют. Однако они могут реагировать на наносимые им удары и, когда игрок атакует их, начинают взрываться. Повтор-

ные взрывы можно производить до тех пор, пока степень повреждения аванпоста не достигнет такого уровня, что он будет окончательно разрушен!

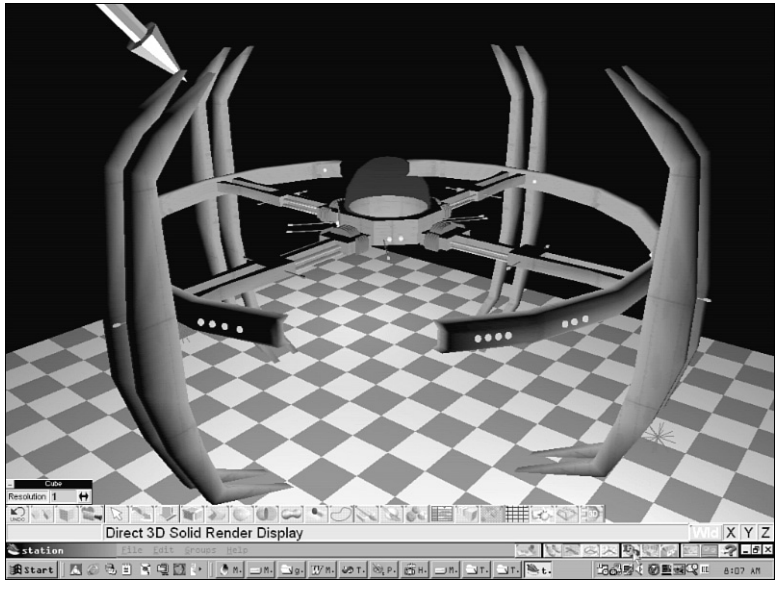


Рис. 15.8. Окончательный вид аванпоста

Мины

Мины являются защитниками аванпостов. Они располагаются вблизи последних до тех пор, пока вы не приблизитесь к аванпосту на определенное расстояние, после чего приходят в действие и атакуют вас. Мины проектировались с помощью TS4 (рис. 15.9 и 15.10).

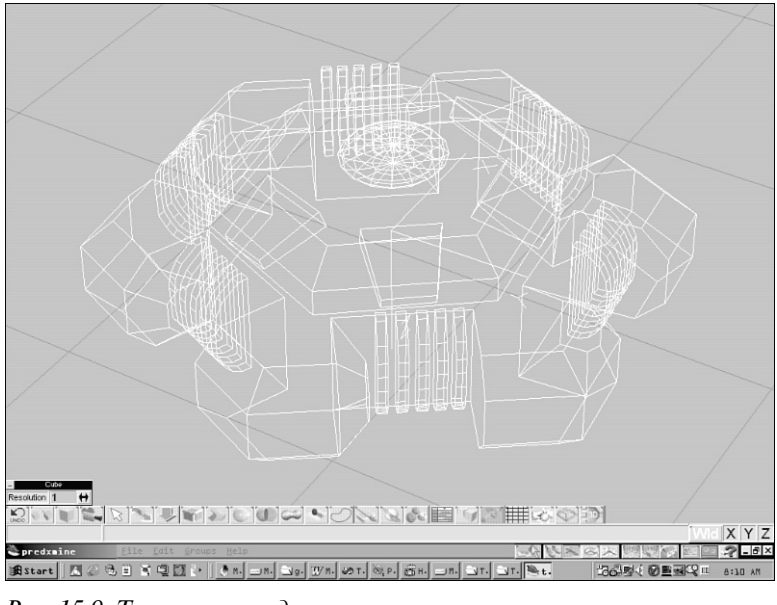


Рис. 15.9. Трехмерная модель мины

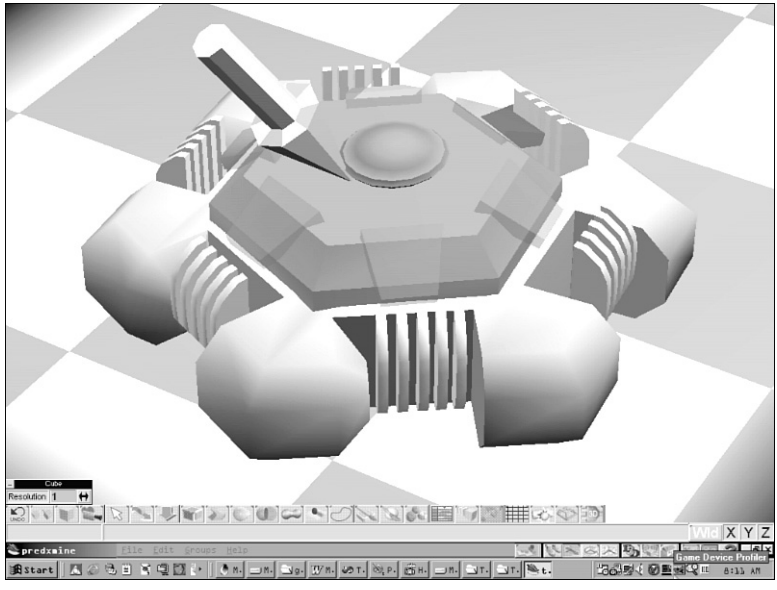


Рис. 15.10. Окончательный вид мины

Я был не особенно доволен, окончив работу с трехмерной моделью мины. Изначально я создавал другую трехмерную модель, показанную на рис. 15.11, но она оказалась больше похожа на стационарную мину, а не на движущуюся, которая способна атаковать вас.

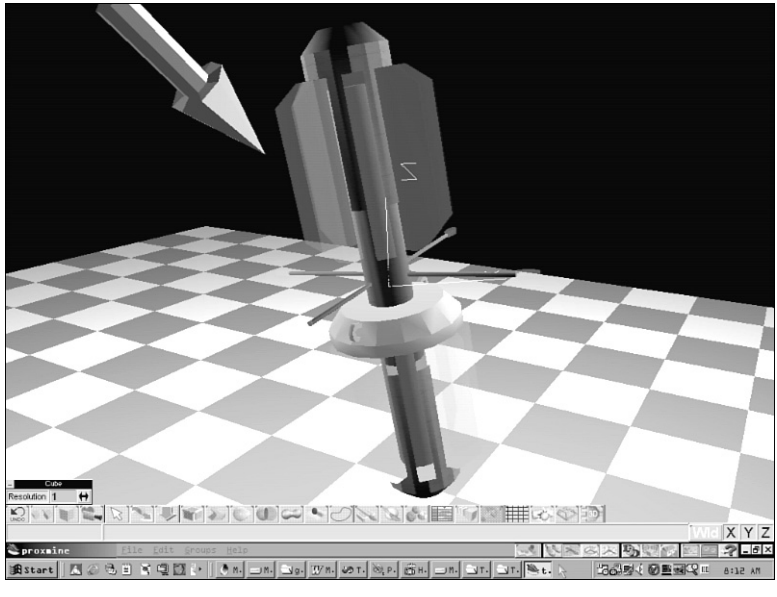


Рис. 15.11. Другая концепция мины

Искусственный интеллект мины примитивен. Это конечный автомат, который начинает работу из *ждущего* или *спящего* состояния. Мина активизируется, когда вы приближаетесь к аванпосту на определенное расстояние, заданное следующей директивой `#define`:

```
#define MIN_MINE_ACTIVATION_DIST 250
```

Если игрок находится в области действия мины, она активизируется и направляется к игроку, используя алгоритм вектора отслеживания, который я описал и продемонстрировал в главе 12, "Искусственный интеллект в игровых программах".

Мины не имеют какого-либо оружия; они просто направляются к вам и взрываются возле вас, нанося повреждения вашему кораблю.

Боевые корабли

Боевые корабли смоделированы с использованием TS4 (рис. 15.12 и 15.13). Они имели множество разных деталей и выглядели просто великолепно, пока я не привел их к окончательному виду, преобразовав в изображение в 8-битовом режиме. Но такова жизнь...

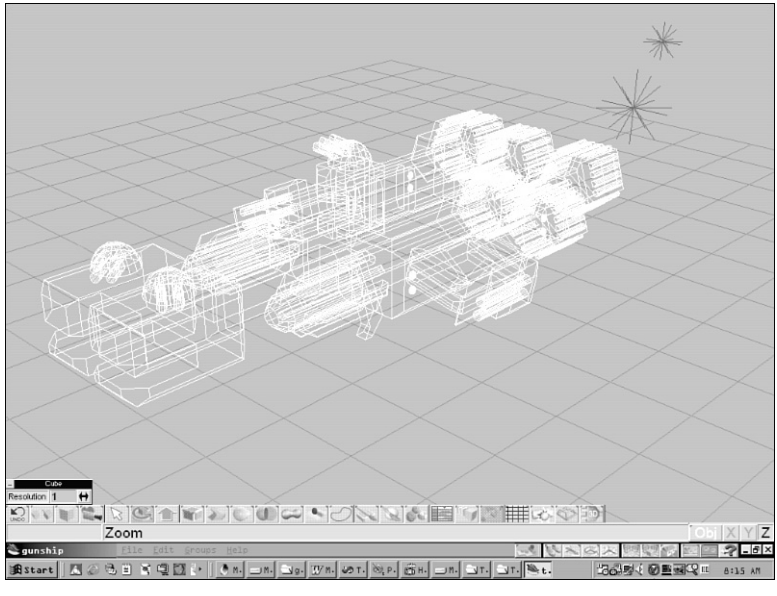


Рис. 15.12. Трехмерная модель боевого корабля

Искусственный интеллект боевых кораблей тоже весьма прост. Они перемещаются по оси X с постоянной скоростью и, оказываясь на определенном расстоянии от игрока, изменяют свою позицию по оси Y для отслеживания игрока, но делают это медленно, так что игрок всегда может успеть изменить направление своего движения и уйти в сторону. Сила боевых кораблей заключена в их мощном вооружении. Каждый боевой корабль оснащен тремя лазерными пушками, которые могут стрелять независимо друг от друга, как показано на рис. 15.14.

Алгоритм отслеживания у пушек отличается некоторой новизной. Алгоритм решает задачу минимизации дистанции от головной части орудийной башни до корабля игрока, для чего вычисляет, к какому результату приведет вращение по часовой стрелке и против нее, т.е. в каком направлении следует производить вращение, чтобы дистанция действительно уменьшилась. После этого выполняется вращение орудийной башни в нужном направлении.

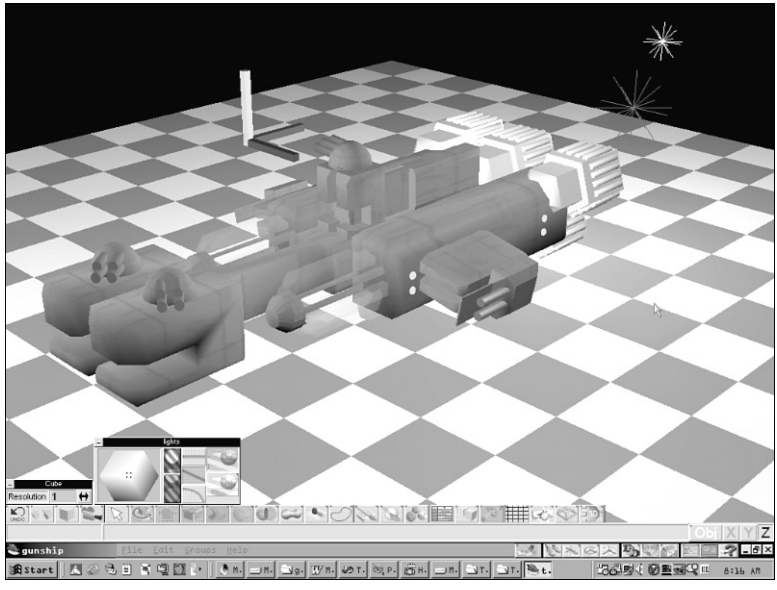


Рис. 15.13. Окончательный вид боевого корабля

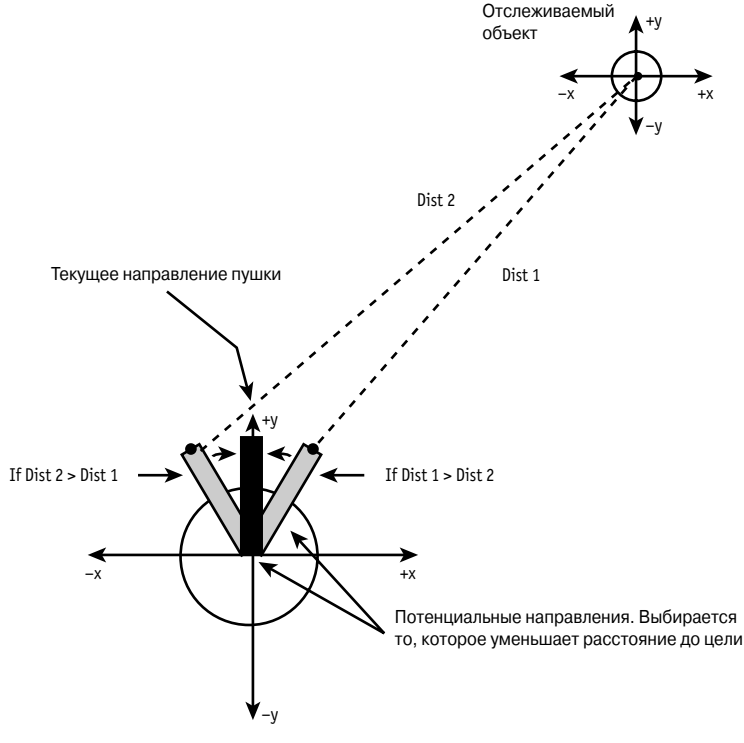


Рис. 15.14. Алгоритм орудийной башни боевого корабля

Алгоритм написан без каких-либо ухищрений и без каких-либо вычислений сложных векторов — использовались только вычисления дистанции и алгоритм минимизации. Я пришел к этой мысли, рассмотрев отслеживание объектов человеком. Мы начинаем вращение головы в направлении объекта и когда чувствуем, что уже видим его краем глаза, начинаем замедлять поворот головы и через некоторое время прекращаем его вовсе. Но иногда мы можем "проскочить" нужное направление, и тогда приходится исправлять положение, поворачивая голову в обратном направлении. Эта идея заложена и при составлении алгоритма. Далее приведен фрагмент кода, в котором записан данный механизм отслеживания.

```
// Создаем вектор, указывающий направление пушки

// Вычисляем вектор
int tdir1 = gunships[index].varsI[INDEX_GUNSHIP_TURRET];

float d1x = gunships[index].varsI[INDEX_WORLD_X]
        + cos_look16[tdir1]*32;
float d1y = gunships[index].varsI[INDEX_WORLD_Y]
        + sin_look16[tdir1]*32;

// Вычисляем правый вектор
int tdir2 = gunships[index].varsI[INDEX_GUNSHIP_TURRET]+1;

if (tdir2 > 15)
   tdir2 = 0;

float d2x = gunships[index].varsI[INDEX_WORLD_X]
        + cos_look16[tdir2]*32;
float d2y = gunships[index].varsI[INDEX_WORLD_Y]
        + sin_look16[tdir2]*32;

// Вычисляем левый вектор
int tdir0 = gunships[index].varsI[INDEX_GUNSHIP_TURRET]-1;

if (tdir0 < 0)
   tdir0=15;

float d0x = gunships[index].varsI[INDEX_WORLD_X]
        + cos_look16[tdir0]*32;
float d0y = gunships[index].varsI[INDEX_WORLD_Y]
        + sin_look16[tdir0]*32;

// Ищем минимальное расстояние
float dist0 = Fast_Distance_2D(player_x - d0x,
                    player_y - d0y);
float dist1 = Fast_Distance_2D(player_x - d1x,
                    player_y - d1y);
float dist2 = Fast_Distance_2D(player_x - d2x,
                    player_y - d2y);

if (dist0 < dist2 && dist0 < dist1)
{
   // Движение влево лучше
   gunships[index].varsI[INDEX_GUNSHIP_TURRET] = tdir0;
```

```
} // if
else
   if (dist2 < dist0 && dist2 < dist1)
   {
      // Движение вправо лучше
      gunships[index].varsI[INDEX_GUNSHIP_TURRET] = tdir2;

   } // if
```

СОВЕТ

Обратите внимание на то, что я использовал несколько вычислений дистанции. Все они используют функцию `Fast_Distance2D()`, так что выполняются очень быстро и представляют собой всего лишь несколько сдвигов и сложений.

Запасы энергии

Я чувствовал некоторую "стратегическую неполноценность" игры из-за наличия бесчисленных боеприпасов и средств защиты, поэтому и подумал, почему бы не иметь некоторое количество летающих в космосе запасов энергии? С этим намерением я засел за TS4 и начал их моделировать. Но я понимал, что пока еще не имею подходящей идеи.

Запасы энергии обычно выглядят не очень реалистично. Я имею в виду то, что, хотя они и называются AMMO (боеприпасы) и обеспечивают работу каких-то антигравитационных двигателей и пушек, все же они должны выглядеть как-то более правильно. В конце концов я принял решение придать запасам вид простых сфер со словами AMMO и SHLD (что означает соответственно "боеприпасы" и "средства защиты"), ярко светящимися внутри сфер. Соответствующие модели показаны на рис. 15.15.

Рис. 15.15. Окончательный вид запасов энергии

Когда трехмерные модели были готовы, я создал изображения запасов и занялся вводом их в программу игры. Я хотел, чтобы запасы появлялись в результате разрушения либо астероида,

либо противника, и в конце концов решил, что имеет смысл разместить их внутри астероидов. Аргументом в пользу этого было то, что при разрушении астероида обломки пород и полезные минералы могут выбрасываться взрывом в окружающее пространство.

При разрушении астероида с некоторой степенью вероятности образуются запасы энергии (боеприпасы или средства защиты), которые медленно уходят с места взрыва. Если вы перемещаетесь сквозь запасы, то поглощаете их, тем самым увеличивая свои средства защиты или боеприпасы.

Очень скоро я понял, что найти потерянные в очень большом пространстве игры запасы будет весьма сложно. Конечно, я могу воспользоваться сканером, но чтобы сделать игру более драматичной, я решил использовать определенное время жизни запасов. Таким образом, через 3–9 секунд не подобранные запасы исчезают.

Пилотажный дисплей

Пилотажный дисплей игры *Outpost* состоит из двух главных компонентов: сканера и набора некоторой тактической информации, включая сведения о скорости, наличии топлива, боеприпасов и средств защиты, о количестве оставшихся кораблей и о количестве набранных игроком очков. Все это показано на рис. 15.16. Для отображения я использовал нежный зеленый цвет. Тактическая информация представляет собой простой GDI-текст, а сканер является набором линий, битовых изображений и пикселей, созданных с помощью DirectX.

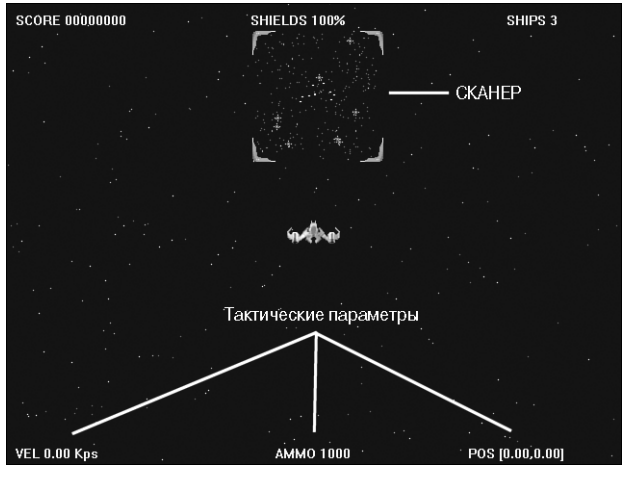


Рис. 15.16. Пилотажный дисплей

Вот как выглядит код сканера:

```
void Draw_Scanner(void)
{
  // Вывод сканера

  int index,sx,sy; // Индекс цикла и положение

  // Блокировка задней поверхности
  DDraw_Lock_Back_Surface();

  // Вывод всех астероидов
```

```
for (index=0; index < MAX_ROCKS; index++)
{
  if (rocks[index].state==ROCK_STATE_ON)
  {
    sx = ((rocks[index].varsI[INDEX_WORLD_X]
      - UNIVERSE_MIN_X) >> 7) + (SCREEN_WIDTH/2)
      - ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
    sy = ((rocks[index].varsI[INDEX_WORLD_Y]
      - UNIVERSE_MIN_Y) >> 7) + 32;

    Draw_Pixel(sx,sy,8,back_buffer, back_lpitch);
  } // if

} // for index


// Вывод боевых кораблей
for (index=0; index < MAX_GUNSHIPS; index++)
{
  // draw gunship blips
  if (gunships[index].state==GUNSHIP_STATE_ALIVE)
  {
    sx = ((gunships[index].varsI[INDEX_WORLD_X]
      - UNIVERSE_MIN_X) >> 7) + (SCREEN_WIDTH/2)
      - ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
    sy = ((gunships[index].varsI[INDEX_WORLD_Y]
      - UNIVERSE_MIN_Y) >> 7) + 32;

    Draw_Pixel(sx,sy,14,back_buffer, back_lpitch);
    Draw_Pixel(sx+1,sy,14,back_buffer, back_lpitch);

  } // if

} // for index

// Вывод мин
for (index=0; index < MAX_MINES; index++)
{
  if (mines[index].state==MINE_STATE_ALIVE)
  {
    sx = ((mines[index].varsI[INDEX_WORLD_X]
      - UNIVERSE_MIN_X) >> 7) + (SCREEN_WIDTH/2)
      - ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
    sy = ((mines[index].varsI[INDEX_WORLD_Y]
      - UNIVERSE_MIN_Y) >> 7) + 32;

    Draw_Pixel(sx,sy,12,back_buffer, back_lpitch);
    Draw_Pixel(sx,sy+1,12,back_buffer, back_lpitch);

  } // if

} // for index
```

```
// Разблокирование задней поверхности
DDraw_Unlock_Back_Surface();

// Вывод всех станций
for (index=0; index < MAX_STATIONS; index++)
{
   if (stations[index].state==STATION_STATE_ALIVE)
   {
      sx = ((stations[index].varsI[INDEX_WORLD_X]
         - UNIVERSE_MIN_X) >> 7) + (SCREEN_WIDTH/2)
         - ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
      sy = ((stations[index].varsI[INDEX_WORLD_Y]
         - UNIVERSE_MIN_Y) >> 7) + 32;

      // Проверка состояния
      if (stations[index].anim_state ==
         STATION_SHIELDS_ANIM_ON)
      {
         stationsmall.curr_frame = 0;
         stationsmall.x = sx - 3;
         stationsmall.y = sy - 3;
         Draw_BOB(&stationsmall,lpddsback);


      } // if
      else
      {

         stationsmall.curr_frame = 1;
         stationsmall.x = sx - 3;
         stationsmall.y = sy - 3;
         Draw_BOB(&stationsmall,lpddsback);


      } // if

   } // if

} // for index


// Блокирование вторичной поверхности
DDraw_Lock_Back_Surface();

sx = ((int(player_x) - UNIVERSE_MIN_X) >> 7)
   + (SCREEN_WIDTH/2) - ((UNIVERSE_MAX_X
   - UNIVERSE_MIN_X) >> 8);
sy = ((int(player_y) - UNIVERSE_MIN_Y) >> 7) + 32;

int col = rand()%256;

Draw_Pixel(sx,sy,col,back_buffer, back_lpitch);
Draw_Pixel(sx+1,sy,col,back_buffer, back_lpitch);
Draw_Pixel(sx,sy+1,col,back_buffer, back_lpitch);
Draw_Pixel(sx+1,sy+1,col,back_buffer, back_lpitch);
```

```
// Разблокирование вторичной поверхности
DDraw_Unlock_Back_Surface();

hud.x       = 320-64;
hud.y       = 32-4;
hud.curr_frame = 0;
Draw_BOB(&hud,lpddsback);

hud.x       = 320+64-16;
hud.y       = 32-4;
hud.curr_frame = 1;
Draw_BOB(&hud,lpddsback);

hud.x       = 320-64;
hud.y       = 32+128-20;
hud.curr_frame = 2;
Draw_BOB(&hud,lpddsback);


hud.x       = 320+64-16;
hud.y       = 32+128-20;
hud.curr_frame = 3;
Draw_BOB(&hud,lpddsback);


} // Draw_Scanner
```

Я показал вам типичную функцию сканера, хотя на самом деле эти функции обычно выглядят более беспорядочно. По существу, сканер показывает позицию всех объектов игры. Проблема состоит в том, как изобразить все огромное пространство игры в небольшом по размеру пространстве сканера и как в нем преобразовать объекты, чтобы они выглядели реалистично, что в принципе невозможно. Поэтому все изображения в сканере имеют одинаковый вид.

Кроме того, вам, глядя на сканер, нужно быстро выбрать необходимую информацию о том, где, например, находитесь вы или где находятся противники и т.д., поэтому цвет и форма объектов очень важны. В конце концов я решил использовать для отображения противников один или несколько пикселей, для отображения астероидов — единичные серые пиксели, а для отображения аванпостов — битовые изображения. Корабль игрока отображается, как маленький ярко светящийся шарик.

Предполагается, что сканер представляет собой некую голографическую систему отображения. Поэтому я захотел, чтобы он выглядел как-нибудь покрасивее, и нарисовал ему изящные уголки.

Чтобы понять, как работает сканер, взгляните на его код. В его функции входит только определение позиций каждого объекта и вывод полученных сведений в окно сканера.

Система частиц

Система частиц игры *Outpost* точно такая же, как описано в главе 13, “Основы физического моделирования”. Частицы могут создаваться с разными скоростями и цветами, и существуют функции для задания параметров частиц, имеющих специфические свойства для имитации взрывов или каких-либо иных эффектов. Важно не то, как частицы действуют (вы об этом и так уже знаете), а как я использовал их.

В игре *Outpost* частицы используются для многих целей. Я хотел, чтобы все противники оставляли после своего уничтожения газовое облако. Я также хотел, чтобы корабль игрока при движении оставлял позади себя плазменный след. И наконец, я хотел, чтобы при повреждении или взрыве какого-либо объекта картина взрыва, созданная с помощью анимации, дополнялась еще и изображением облаков, образованных большим количеством частиц.

Новизна идеи об использовании частиц состоит в том, что при всей простоте этого средства визуальный эффект от его использования получается очень сильный. Излишне напоминать, что газовые следы и частицы могут быть использованы как самостоятельные элементы игры.

Процесс игры

Игра в *Outpost* очень проста: вам нужно только летать вокруг и взрывать разные объекты. Тем не менее, если сформулировать цель игры, то она будет такой: взорвать все аванпосты.

Ниже перечислены клавиши управления действиями корабля.

<←>, <→>	Поворот корабля
<↑>	Движение вперед
<Ctrl>, <Spacebar>	Стрельба
<H>	Отображение тактической информации
<S>	Отображение сканера
<Left Alt+Right Alt+A>	Специальная комбинация
<Esc>	Выход из игры

Компиляция игры *Outpost*

Компиляция игры *Outpost* не отличается от компиляции других демонстрационных программ, которые вы уже создавали ранее. Рис. 15.17 показывает компоненты, необходимые для компиляции и запуска проекта.

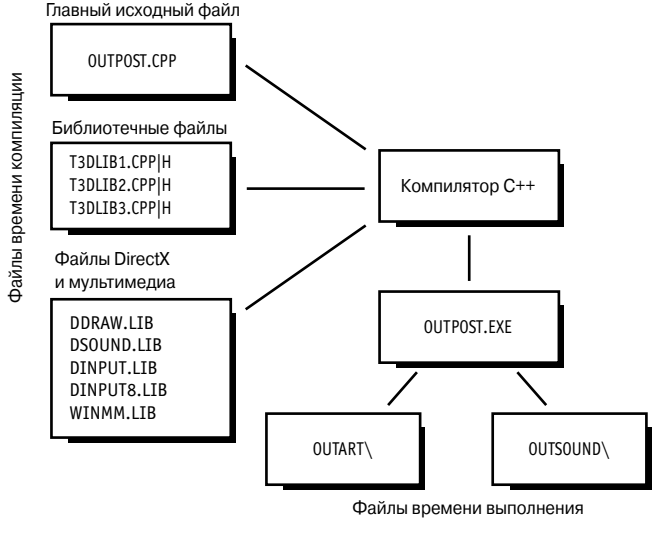


Рис. 15.17. Архитектура программы игры Outpost

Компилируемые файлы

Исходные файлы	
`OUTPOST.CPP`	Основной файл игры
`T3DLIB1.CPP|H`	Первая часть библиотеки игры
`T3DLIB2.CPP|H`	Вторая часть библиотеки игры
`T3DLIB3.CPP|H`	Третья часть библиотеки игры

Библиотечные файлы	
`DDRAW.LIB`	MS DirectDraw
`DSOUND.LIB`	MS DirectSound
`DINPUT.LIB` и `DINPUT8.LIB`	MS DirectInput
`WINMM.LIB`	Библиотека расширений Win32 Multimedia

СОВЕТ

В свой проект вы должны включить библиотечные файлы DirectX; указания пути поиска будет недостаточно. Кроме того, вы должны указать путь поиска заголовочных файлов DirectX в инсталляционном каталоге DirectX SDK.

Файлы времени исполнения

Основной `.EXE`-файл	
`OUTPOST.EXE`	Это основной `.EXE`-файл игры. Он может находиться где угодно, лишь бы имелись соответствующие подкаталоги с ресурсами

Каталоги ресурсов	
`OUTART\`	Каталог изображений для игры
`OUTSOUND\`	Каталог звуковых эффектов для игры

И конечно, вам нужны рабочие файлы DirectX.

Эпилог

Я безмолвствую... Мы сделали все, что могли; конечно, если забыть, что в этой книге ни слова не сказано о трехмерных играх.

Однако на прилагаемом к книге компакт-диске очень много информации о программировании игр вообще и о Direct 3D и General 3D в частности. Кроме того, в приложениях можно найти много полезных сведений по ресурсам, связанным с программированием игр, а также по C++ и математике.

Эта книга оказалась самым сложным проектом, которым я когда-либо занимался. Сначала планировалось сделать трехтомное издание по трехмерному программированию игр под названием *The Necronomicon de Gam*, но потом мы решили сделать одну книгу и дать ей название *Tricks*. Затем, примерно на полпути создания книги, мы поняли, что это неправильно — пытаться втиснуть весь материал в объем менее 2000–3000 страниц, и ограничились книгой по двухмерным играм. Оглядываясь назад, я полагаю, что выбран правильный путь. Все, что я хотел сказать в этой книге, нашло место в ней без всяких сокращений. :)

Хотя я и не имею каких-либо конкретных планов, могу сказать, что есть еще две большие темы, которые могли бы подвигнуть меня на работу над новой книгой: сетевые игры и программирование консольных игр. Кроме того, я думаю, было бы весьма неплохо обратиться к теме программирования для PlayStation I и II, Dreamcast, Xbox и Game Boy Advance.

Впрочем, возможно, вы больше не увидите ничего написанного мной, потому что это занятие очень меня измучило. Я работал над этой книгой по 120 часов в неделю беспрерывно на протяжении целого года. В конце концов меня утешает только мысль о том, сколько новых составителей игровых программ впервые увидят свечение экрана и движущиеся по нему маленькие изображения. Это действительно единственное реальное удовлетворение, которое я получаю от своей работы.

В заключение позволю себе дать вам один совет:

Судьба обратит на вас внимание только тогда, когда вы будете прилагать максимум усилий на пути к своей вершине. В мире нет ничего такого, что было бы невозможным. Если вы поверили, что можете что-то сделать, — вы действительно можете это сделать. Поэтому — вперед!

ЧАСТЬ IV

Приложения

Приложение А

Содержание компакт-диска 827

Приложение Б

Установка DirectX и использование компилятора C/C++ 829

Приложение В

Обзор математики и тригонометрии 833

Приложение Г

Азы C++ 845

Приложение Д

Ресурсы по программированию игр 865

ПРИЛОЖЕНИЕ А

Содержание компакт-диска

Прилагаемый компакт-диск содержит исходный код, исполняемые файлы, демонстрационные программы, шаблонный иллюстративный материал, различное программное обеспечение, звуковые эффекты, книги в электронном виде, графические редакторы и технические статьи. Ниже приведена структура каталогов диска.

```
CD-DRIVE:\
  T3DGAMER1\
  SOURCE\
    T3DCHAP01\
    T3DCHAP02\
      .
      .
    T3DCHAP14\
    T3DCHAP15\

  APPLICATIONS\
  ARTWORK\
    BITMAPS\
    MODELS\
  SOUND\
    WAVES\
    MIDI\
DIRECT X\
GAMES\
ARTICLES\

ONLINEBOOKS\
ENGINES\
```

Каждый из основных каталогов содержит определенные данные. Ниже приведена более подробная информация о каталогах.

T3DGAMER1	Корневой каталог, который содержит все другие подкаталоги. Прочитайте файл README.TXT, в котором детально описаны последние изменения
SOURCE	Содержит все каталоги с исходными текстами программ по главам книги. Скопируйте каталог SOURCE\ на ваш жесткий диск и работайте с ним уже на новом месте
APPLICATIONS	Содержит демонстрационные программы, которые различные компании любезно разрешили поместить на данный компакт-диск
ARTWORK	Содержит шаблонный иллюстративный материал, который можно бесплатно использовать при создании игр
SOUND	Содержит шаблонные звуковые эффекты и музыку, которые можно бесплатно использовать при создании игр
DIRECTX	Содержит самую последнюю версию DirectX SDK
GAMES	Содержит ряд условно-бесплатных двухмерных (2D) и трехмерных (3D) игр, которые, на мой взгляд, вам понравятся
ARTICLES	Содержит полезные для вашего образования статьи, написанные ведущими специалистами в области программирования игр
ONLINEBOOKS	Содержит две электронные книги, посвященные трехмерной графике вообще и Direct3D в частности
ENGINES	Содержит ряд двухмерных (2D) и трехмерных (3D) процессоров, включая такие, как Genesis 3D (с полной документацией) и PowerRender

Поскольку на компакт-диске огромное количество разных программ и данных, общей программы по установке содержимого компакт-диска нет. Процесс инсталляции я оставляю на ваше усмотрение. Необходимо только скопировать каталог SOURCE\ на жесткий диск. Что касается всех других программ и данных, их можно устанавливать по мере необходимости. Просто перенесите их на свой жесткий диск и запустите различные установочные программы, находящиеся в каждом каталоге.

ВНИМАНИЕ

При копировании файлов с компакт-диска устанавливаются биты ARCHIVE и READ-ONLY. Убедитесь, что эти биты у файлов, которые вы копируете на жесткий диск, сброшены. Вы можете сбросить биты в Windows с помощью выбора файлов или каталога; используйте набор клавиш <Ctrl+A> для выделения всех файлов, щелкните правой кнопкой мыши и выберите в контекстном меню пункт File Properties. После открытия диалогового окна Properties сбросьте биты READ-ONLY и ARCHIVE, а затем для завершения работы щелкните на кнопке Apply.

ПРИЛОЖЕНИЕ Б

Установка DirectX и использование компилятора C/C++

К наиболее важной части компакт-диска относятся DirectX SDK и файлы времени исполнения, которые следует инсталлировать. Программа установки находится в каталоге вместе с файлом REDME.TXT, в котором даны пояснения по поводу последних внесенных изменений.

C++

Чтобы иметь возможность работать с этой книгой и компакт-диском, на вашем компьютере должна быть установлена версия DirectX 8.0 SDK или выше. Если вы не уверены, что имеете последнюю версию, запустите инсталляционную программу, во время работы которой вы сможете узнать о текущей версии DirectX SDK.

При установке DirectX SDK обратите внимание на то, куда программа инсталляции поместит файлы SDK. Если у вас появится желание самостоятельно компилировать приложения DirectX, необходимо будет указать компилятору пути поиска библиотечных и заголовочных файлов.

Кроме того, при установке DirectX SDK программа инсталляции сделает запрос, хотите ли вы установить файлы DirectX времени выполнения или нет. Для работы программ вам понадобятся как файлы времени выполнения, так и SDK. Существует две версии библиотек времени выполнения.

Debug. Эта версия содержит средства отладки и диагностирования, и именно ее я рекомендую вам установить для разработки приложений. Однако следует учесть, что в этом случае программы DirectX запускаются немного медленнее.

Retail. Это полная версия для потребителей, содержащая все, что ожидает от нее рядовой потребитель. Она работает быстрее, чем отладочная версия. При желании вы можете со временем установить ее поверх уже установленной отладочной версии.

C++

Для пользователей программных продуктов компании Borland (если таковые еще остались): DirectX SDK содержит библиотеки импорта DirectX. Их можно найти в каталоге `BORLAND\` после установки DirectX SDK на ваш компьютер и использовать при компиляции. Зайдите на Web-узел Borland, а также ознакомьтесь с файлом `README.TXT` в каталоге `BORLAND\`, чтобы узнать о последних рекомендациях по компилированию программ DirectX с помощью компиляторов Borland.

И наконец, к тому времени, как я закончил писать эту книгу, компания Microsoft выпустила несколько новых версий DirectX. Время от времени обновляйте DirectX SDK, заходя на Web-узел DirectX по адресу: `http://www.microsoft.com/directx/`.

Использование компилятора C/C++

В течение последних трех лет я получил более 17 000 сообщений по электронной почте от людей, которые не знают, как использовать компилятор C/C++. Я не хочу больше отвечать на письма о проблемах с компилятором, если, конечно, компьютер вдруг не заговорит человеческим языком! Каждая из присланных мне проблем возникала при работе с компилятором пользователя-новичка. Не стоит пытаться использовать такое сложное программное обеспечение, как компилятор C/C++, не прочитав руководство! Или вы не согласны со мной? Поэтому, прежде чем компилировать программы из этой книги, прочитайте документацию, которой снабжен ваш компилятор.

Что касается компиляции, то при работе над книгой я использовал MS VC++ 5.0 и 6.0, и с этими компиляторами все работало отлично. Думаю, что VC++ 4.0 также должен работать, но не настаиваю на этом (я слышал, что у VC++ 4.0 существуют некоторые проблемы с DirectX). Если у вас установлены компиляторы Borland или Watcom, они также должны работать, но для компилирования, возможно, придется немного повозиться с их настройками. Чтобы избавиться от головной боли, советую приобрести копию VC++. Версии Student и Standard обычно стоят менее 100 долларов.

Подводя итоги, стоит отметить, что компилятор Microsoft является лучшим для программ Windows/DirectX. Я использую компиляторы Borland и Watcom для других целей, но для приложений Windows; однако я не знаю профессиональных программистов игр, которые не используют MS VC++. Это подходящее инструментальное средство для успешной работы (примечание для Билла Гейтса: мой номер счета в банке Кайман — 412-0300686-21; заранее благодарен за оплату рекламы).

Ниже приведены несколько рекомендаций по настройке компиляторов MS VC++. Другие компиляторы настраиваются аналогично.

Тип приложений. Программы DirectX являются программами Windows, а точнее, `.EXE`-приложениями Win32. Поэтому настройте компилятор для создания `.EXE`-приложений Win32 для *всех* программ DirectX. Если вы работаете над *консольным* приложением, то настройте компилятор для создания *консольных* приложений.

Каталоги поиска. Чтобы компилировать программы DirectX, компилятору нужны библиотечные (`.LIB`) и заголовочные (`.H`) файлы. Для того чтобы компилятор мог найти необходимые библиотечные и заголовочные файлы DirectX SDK, пути к ним должны быть указаны в настройках компилятора. Однако этого недостаточно! Вы должны убедиться, что пути к файлам DirectX стоят *первыми* в списке поиска. Причина в том, что VC++ поставляется со старой версией DirectX и именно эти файлы будет находить ваш

компилятор. Кроме того, убедитесь, что вы *вручную* включили .LIB-файлы DirectX в ваши проекты. В проекты всегда следует включать такие библиотеки, как DDRAW.LIB, DSOUND.LIB, DINPUT.LIB, DINPUT8.LIB и другие. Это очень важно!

Установка уровня ошибки. Убедитесь, что уровень ошибки в компиляторе установлен в разумных пределах, например уровень 1 или 2. Не отключайте сообщения об ошибках, но и не снимайте на них ограничения. Код в этой книге профессионального уровня, и компилятор будет считать, что в нем имеется множество "подозрительных" мест. Поэтому снизьте уровень предупреждений.

Ошибки приведения типов. Если компилятор выдает ошибку приведения типов в строке кода (пользователи VC++ 6.0 должны быть осторожны), нужно просто сделать явное преобразование! Я получил более 3000 электронных писем от людей, которые не знают, что такое приведение типов. Если вы не знаете, что это такое, обратитесь к книге по C/C++. Должен заметить, что в моем коде вполне может быть пропущено явное приведение типов. Если вы получите одну из таких ошибок, посмотрите, что именно от вас ожидает компилятор, и просто поместите оператор преобразования к данному типу перед rvalue.

Настройки оптимизации. До тех пор пока вы не создаете окончательную версию продукта, не позволяйте компилятору оптимизировать код. Установите не более чем стандартный уровень оптимизации, отдавая предпочтение скорости работы приложения, а не его размеру.

Модели потоков. В этой книге 99% примеров однопоточны, поэтому используйте однопоточные библиотеки. Если вам незнаком этот термин, прочитайте документацию на компилятор. Когда в книге используются многопоточные библиотеки, я обязательно сообщаю об этом. Такая программа имеется, например, в главе 11, "Алгоритмы, структуры данных, управление памятью и многопоточность", где явно сказано, что вы должны использовать многопоточные библиотеки.

Генерация кода. Тонкая настройка кода, генерируемого компилятором. Поскольку процессоры i486, похоже, ушли в прошлое, установите настройку генерации кода для процессоров типа Pentium.

Выравнивание структур. Управляет "заполнением" полей структур. Процессоры Pentium лучше работают с данными, расположенными в памяти по адресам, которые кратны 32 байтам, поэтому установите выравнивание как можно более высоким. Это приведет к увеличению кода и потребности в памяти, но программа при этом станет работать быстрее.

И наконец, когда вы компилируете программы, убедитесь, что включили все исходные файлы, к которым обращается главная программа. Например, если вы видите, что в текст включен заголовочный файл T3DLIB1.H, то наверняка в проект следует включить и файл T3DLIB1.CPP.

ПРИЛОЖЕНИЕ В

Обзор математики и тригонометрии

Я люблю математику. А знаете почему? Потому что она не обсуждаема. Мне не нужно думать, как сделать что-то наилучшим способом. Этот способ, если он есть, единственный.

Этот небольшой математический обзор разбит на разделы, поэтому можно что-то пропустить, перескочить от одного раздела к другому. Обзор, по сути, напоминает небольшой справочник.

Тригонометрия

Это раздел математики, изучающий соотношения между сторонами и углами треугольника. Большинство тригонометрических соотношений выведены из анализа прямоугольного треугольника, как показано на рис. В.1.

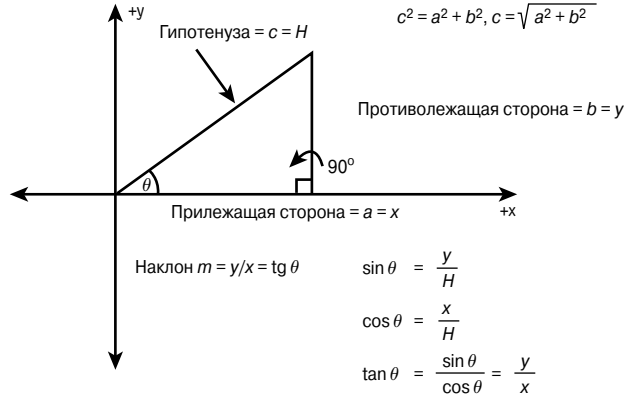


Рис. В.1. Прямоугольный треугольник

В табл. В.1 приведены значения углов в радианах и градусах (далее все углы будут выражаться в радианах, если только явно не оговорено иное).

Таблица В.1. Соотношение радианов и градусов

$360^o = 2\pi$ радиан или приблизительно 6,28 радиан
$180^o = \pi$ радиан или приблизительно 3,14159 радиан
360^o равно 2π радиан, откуда 1 радиан приблизительно равен 57^o,296
2π радиан равно 360^o, откуда 1^o примерно равен 0,0175 радиан

Ниже приведены некоторые тригонометрические утверждения.

Утверждение 1. Полная окружность составляет 2π радиан. Функции математической библиотеки sin() и cos() принимают параметры, выраженные в радианах, а не в градусах — не забывайте об этом! Еще раз вернитесь к табл. В.1.

Утверждение 2. Сумма внутренних углов треугольника $\theta_1 + \theta_2 + \theta_3 = \pi$.

Утверждение 3. В треугольнике (см. рис. В.1) сторону, противоположную углу θ_1, называют *противолежащей стороной*, сторону, прилегающую к углу θ_1, — *прилежащей стороной*. В прямоугольном треугольнике прилежащие к прямому углу стороны называются *катетами*, а противолежащая — *гипотенузой*.

Утверждение 4. Сумма квадратов катетов прямоугольного треугольника равна квадрату гипотенузы. Это утверждение называют *теоремой Пифагора*.

Следовательно, зная две стороны прямоугольного треугольника, можно легко определить третью сторону.

Утверждение 5. Математики часто используют такие главные тригонометрические функции, как *синус*, *косинус* и *тангенс*, определяемые следующим образом:

$$\sin\theta = \frac{y}{r},\ 0 \le \theta \le 2\pi,\ -1 \le \sin \le 1;$$

$$\cos\theta = \frac{x}{r},\ 0 \le \theta \le 2\pi,\ -1 \le \cos \le 1;$$

$$\mathrm{tg}\,\theta = \frac{\sin\theta}{\cos\theta} = \frac{y}{x},\ -\frac{\pi}{2} \le \theta \le \frac{\pi}{2},\ -\infty < \mathrm{tg} < +\infty.$$

На рис. В.2 показаны графики всех описанных тригонометрических функций. Обратите внимание на *периодичность* этих функций. Функции sin() и cos() имеют период 2π, в то время как tan()[1] имеет период π. Обратите также внимание на разрывность функции tan(), когда ее аргумент становится равным $\pi/2$ по модулю π.

Теперь, когда основные тригонометрические тождества и соотношения введены, можете взять учебник математики и доказать их. Я же покажу вам ряд соотношений, которые должен знать программист игр. В табл. В.2 приведены некоторые тригонометрические определения, соотношения и тождества.

[1] Функция "тангенс" в математических формулах имеет принятое в отечественной математической литературе обозначение tg; но когда речь идет о программе на языке C/C++, то в математической библиотеке этих языков тангенс обозначается как tan(). — *Прим. ред.*

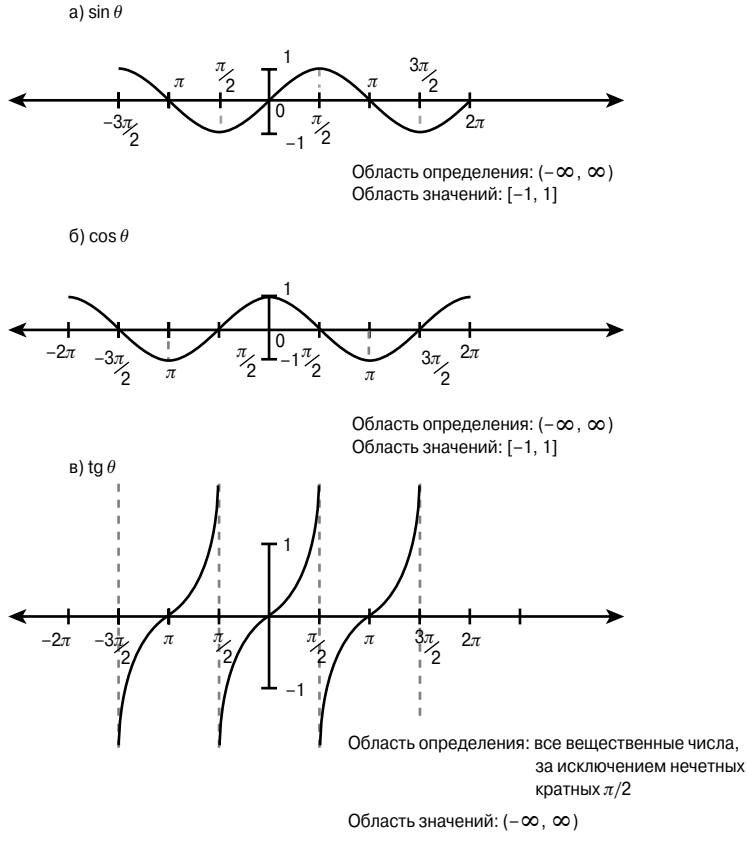


Рис. В.2. Графики основных тригонометрических функций

Таблица В.2. Используемые тригонометрические тождества

Косеканс: $\operatorname{cosec}\theta = 1/\sin\theta$

Секанс: $\sec\theta = 1/\cos\theta$

Котангенс: $\operatorname{ctg}\theta = 1/\operatorname{tg}\theta$

Теорема Пифагора, выраженная через тригонометрические функции:

$$(\sin\theta)^2 + (\cos\theta)^2 = 1$$

Тригонометрические преобразования:

$$\sin\theta = \cos(\theta - \pi/2)$$

$$\sin(-\theta) = -\sin(\theta)$$

$$\cos(-\theta) = \cos(\theta)$$

$$\sin(\theta \pm \phi) = \sin\theta\cos\phi \pm \cos\theta\sin\phi$$

$$\cos(\theta \pm \phi) = \cos\theta\cos\phi \mp \sin\theta\sin\phi$$

Конечно, при желании можно вывести все эти тождества самостоятельно. Тождества упрощают сложные тригонометрические формулы. Поэтому, когда вам придется составлять алгоритмы на основе синуса, косинуса, тангенса и т.д., всегда используйте тригонометрические формулы для упрощения математических уравнений. Помните: скорость, скорость и еще раз скорость!

Векторы

Векторы — это лучшие друзья программиста игр. Вектор представляет собой отрезок прямой, определяемый начальной и конечной точкой, как показано на рис. В.3.

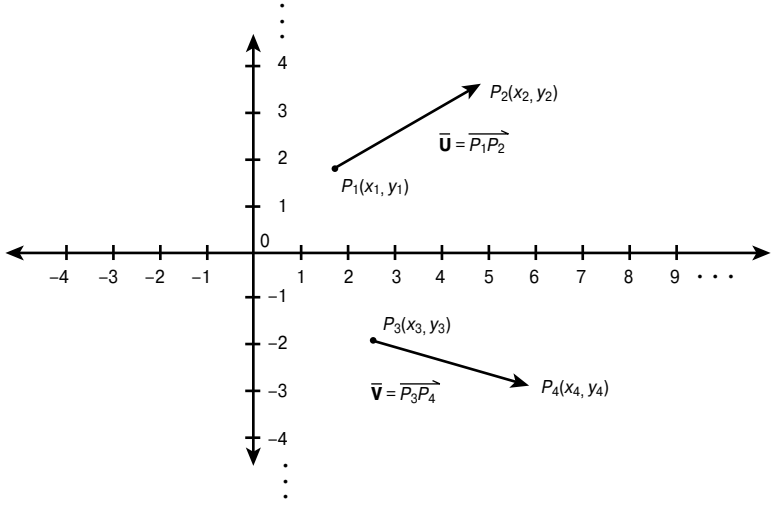


Рис. В.3. Векторы на плоскости

На рис. В.3 показан вектор **U**, определяемый двумя точками: p_1 (начальная точка) и p_2 (конечная точка). Вектор **U**=<u_x,u_y> идет из точки $p_1(x_1,y_1)$ к точке $p_2(x_2,y_2)$. Для вычисления вектора **U** необходимо из координат конечной точки вычесть координаты начальной точки:

$$\mathbf{U} = p_2 - p_1 = \langle x_2 - x_1, y_2 - y_1 \rangle = \langle u_x, u_y \rangle .$$

Обычно векторы обозначают полужирными прописными буквами, например **U**. Компоненты вектора при написании заключают в угловые скобки, т.е. записывают вектор как $\mathbf{U} = \langle u_x, u_y \rangle$.

Итак, вектор — это отрезок прямой, идущий от начальной точки к конечной, и этот отрезок может представлять множество понятий, например *скорость*, *ускорение* и т.п. Внимание: будучи определен, вектор всегда рассматривается относительно начала координат. Это означает, что, после того как вы создали вектор, идущий из точки p_1 в точку p_2, его начальная точка в векторном пространстве всегда имеет координаты $(0,0)$ или $(0,0,0)$ в случае трехмерного пространства.

Вектор определяется двумя (в двухмерном пространстве) или тремя (в трехмерном пространстве) компонентами, т.е. фактически конечной точкой в двух- или трехмерном пространстве (как уже сказано, начальная точка вектора всегда располагается в начале координат). Это не означает, что вы не можете вращать векторы и производить с ними различные геометрические операции. Просто всегда надо помнить, что вектор по определению — величина относительная.

Поскольку векторы являются множествами упорядоченных чисел, над ними можно производить многие обычные математические операции, выполняя эти операции с каждым компонентом вектора независимо.

C++

Векторы могут иметь любое число компонентов. Обычно в компьютерной графике имеют дело с двух- и трехмерными векторами, т.е. векторами типа $\mathbf{A}=<x, y>$, $\mathbf{B}=<x, y, z>$. N-мерный вектор имеет вид $\mathbf{C}=<c_1, c_2, c_3, \ldots, c_n>$. N-мерные векторы обычно используют для представления множеств переменных, а не геометрического пространства, поскольку после трехмерного пространства мы попадаем в фантастический мир гиперпространств.

Длина вектора

Первое, что чаще всего требуется при работе с векторами, — это вычисление их длины. Длина вектора называется *нормой* и обозначается двумя вертикальными черточками: $|\mathbf{U}|$. Это и есть длина вектора **U**.

Длину вектора вычисляют как расстояние от начала координат до конца вектора. Следовательно, для определения длины можно использовать теорему Пифагора:

$$|\mathbf{U}| = \sqrt{u_x^2 + u_y^2} .$$

Если мы имеем дело с трехмерным пространством, то длина вектора равна соответственно $|\mathbf{U}| = \sqrt{u_x^2 + u_y^2 + u_z^2}$.

Нормирование

После того как длина вектора определена, над ним можно выполнить разные интересные действия. Можно нормировать вектор, т.е. сжать так, чтобы его длина стала равна 1. Единичные векторы, как и скаляры, равные 1, обладают множеством замечательных свойств (думаю, в душе вы со мной согласны). Для вектора $\mathbf{N} = \langle n_x, n_y \rangle$ нормированный вектор записывают строчной буквой **n** и вычисляют по формуле: $\mathbf{n} = \mathbf{N} / |\mathbf{N}|$.

Все очень просто. Нормированный вектор — это просто вектор, деленный на собственную длину (ну или умноженный на величину, обратную длине).

Умножение на скаляр

Следующая операция, которую вы, скорее всего, захотите выполнить над вектором, — масштабирование. Эту операцию выполняют путем умножения каждого компонента вектора на скалярное число, например:

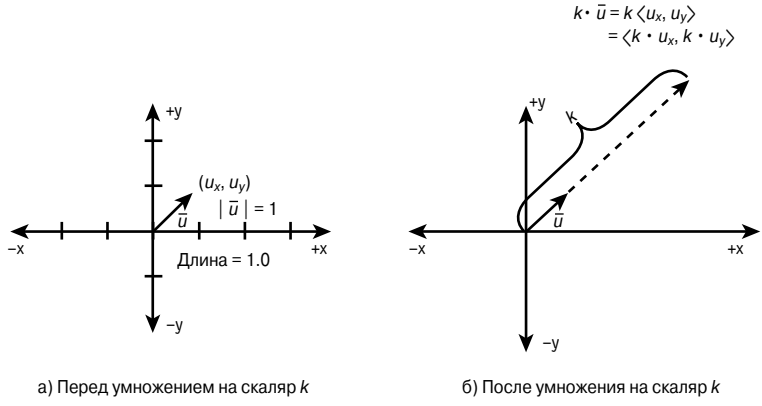
$$\mathbf{U} = \langle u_x, u_y \rangle,$$
$$k\mathbf{U} = k\langle u_x, u_y \rangle = \langle ku_x, ku_y \rangle.$$

На рис. В.4 операция масштабирования показана графически.

Добавим, что для того, чтобы инвертировать направление вектора, можно просто умножить его на -1 (рис. В.5).

Математически это выглядит так:

$$\mathbf{U} = \langle u_x, u_y \rangle,$$
$$-\mathbf{U} = -1\langle u_x, u_y \rangle = \langle -u_x, -u_y \rangle.$$



а) Перед умножением на скаляр *k* б) После умножения на скаляр *k*

Рис. B.4. Масштабирование вектора

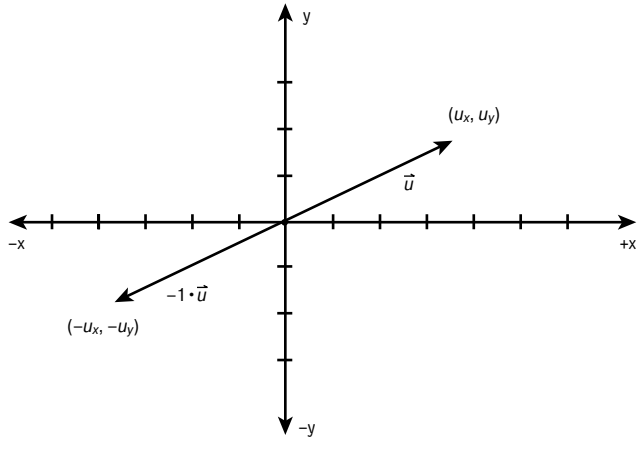


Рис. B.5. Инверсия вектора

Сложение векторов

Для того чтобы сложить два вектора, достаточно сложить соответствующие компоненты векторов. На рис. B.6 операция сложения представлена графически.

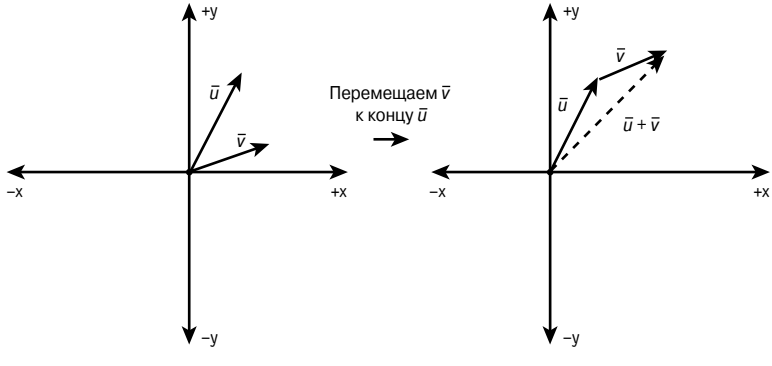


Рис. B.6. Сложение векторов

Вектор **U** прибавляют к вектору **V** и получают вектор **R**. Обратите внимание на геометрическое выполнение этой операции. Для того чтобы сложить векторы **U** и **V**, вектор **V** путем параллельного переноса перемещают в конечную точку вектора **U** и затем соединяют полученную точку с началом координат; образовавшаяся сторона треугольника и будет представлять собой вектор суммы. Геометрически это эквивалентно следующей операции:

$$\mathbf{U}+\mathbf{V}=\langle u_x,u_y\rangle+\langle v_x,v_y\rangle=\langle u_x+v_x,u_y+v_y\rangle.$$

Таким образом, чтобы сложить любое число векторов графически, можно просто сложить их "голова к хвосту". Суммой векторов будет вектор, идущий от начала координат к концу последнего вектора.

Вычитание векторов

Вычитание векторов, по сути, является сложением вектора с вектором с противоположным знаком. Однако иногда полезно рассмотреть графически и вычитание векторов, что и сделано на рис. В.7.

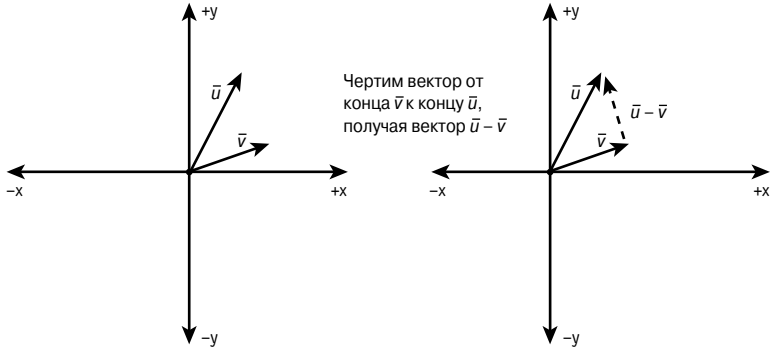


Рис. В.7. Вычитание векторов

Обратите внимание, что **U–V** вычисляют путем построения вектора от **V** к **U**, а **V–U** вычисляют путем построения вектора от **U** к **V**. Математически вычитание векторов записывается так:

$$\mathbf{U}-\mathbf{V}=\langle u_x,u_y\rangle-\langle v_x,v_y\rangle=\langle u_x-v_x,u_y-v_y\rangle.$$

Это выражение легко запомнить, но бумага в клеточку иногда оказывается лучше компьютера, поскольку вы быстрее можете увидеть полученный результат. Поэтому советую вам научиться графически складывать и вычитать векторы при создании алгоритмов — это очень помогает, поверьте мне!

Внутреннее (скалярное) произведение векторов

Пришло время задать вопрос: "Можно ли перемножить два вектора?". Конечно да, но, оказывается, прямой способ перемножения компонентов не слишком полезен. Другими словами,

$$\mathbf{U}\cdot\mathbf{V}=\langle u_x\cdot v_x,\ u_y\cdot v_y\rangle$$

в векторном пространстве не означает ничего. Однако имеет смысл скалярное произведение векторов, определяемое следующей формулой:

$$\mathbf{U}\cdot\mathbf{V}=u_x\cdot v_x+u_y\cdot v_y.$$

Скалярное произведение, обозначаемое точкой (·), обычно вычисляют путем сложения произведений отдельных членов. Результатом такого умножения является скаляр. И что это нам дает? Ведь теперь у нас больше нет векторов. Верно, но точечное произведение, как выясняется, равно следующему выражению:

$$\mathbf{U} \cdot \mathbf{V} = |\mathbf{U}| \cdot |\mathbf{V}| \cdot \cos\theta .$$

В соответствии с этим выражением скалярное произведение **U** и **V** равно произведению длин **U** и **V**, умноженному на косинус угла между ними. Если вы объедините два выражения, то получите способ вычисления угла между двумя векторами:

$$\cos\theta = \frac{u_x v_x + u_y v_y}{\sqrt{\left(u_x^2 + u_y^2\right)\left(v_x^2 + v_y^2\right)}} .$$

Это очень интересная формула, которая дает путь для вычисления угла между двумя векторами, как показано на рис. В.8, и потому чрезвычайно полезна.

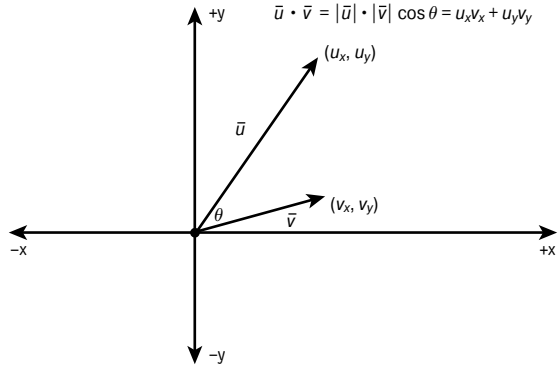


Рис. В.8. Скалярное произведение

Из этой формулы следует несколько интересных фактов.

Факт 1. Если угол между **U** и **V** прямой, то $\mathbf{U} \cdot \mathbf{V} = 0$.
Факт 2. Если угол между **U** и **V** острый, то $\mathbf{U} \cdot \mathbf{V} > 0$.
Факт 3. Если угол между **U** и **V** тупой, то $\mathbf{U} \cdot \mathbf{V} < 0$.
Факт 4. Если **U** и **V** равны, то $\mathbf{U} \cdot \mathbf{V} = |\mathbf{U}|^2 = |\mathbf{V}|^2$.

Эти факты графически представлены на рис. В.9.

Векторное произведение

Следующий тип умножения, который применяют к векторам, называется *векторным произведением*. Векторное произведение имеет смысл только в отношении векторов с тремя или более компонентами, поэтому в качестве примера рассмотрим трехмерные векторы. Пусть $\mathbf{U} = \langle u_x, u_y, u_z \rangle$ и $\mathbf{V} = \langle v_x, v_y, v_z \rangle$. Тогда векторное произведение определяется как

$$\mathbf{U} \times \mathbf{V} = |\mathbf{U}| \cdot |\mathbf{V}| \cdot \sin\theta \cdot \mathbf{n} .$$

Рассмотрим это выражение по частям. Произведение длин векторов на синус угла между ними — $|\mathbf{U}| \cdot |\mathbf{V}| \cdot \sin\theta$ — является скаляром, который можно умножить на **n**. Но что

такое **n**? Это единичный вектор (вот почему он записан строчной буквой), который перпендикулярен и к **U**, и к **V**. На рис. B.10 векторное произведение показано графически.

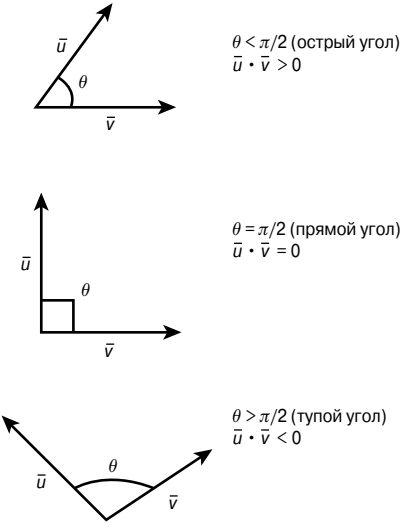


Рис. B.9. Углы и их связь со скалярным произведением

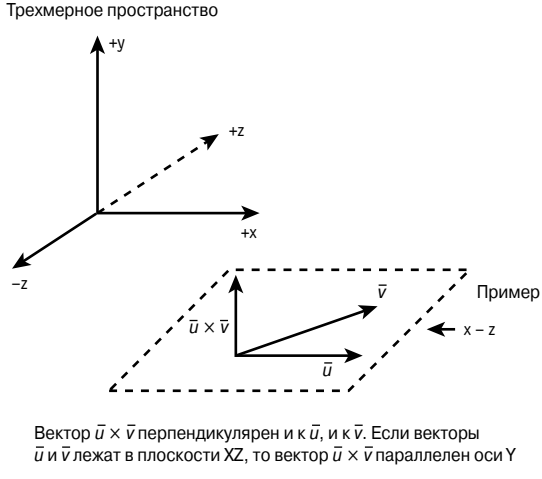


Рис. B.10. Векторное произведение

Итак, векторное произведение дает нам информацию об угле между векторами **U** и **V** и о векторе, перпендикулярном к обоим векторам-множителям. Но мы не в состоянии получить ничего толкового из такого определения. Вопрос в том, как вычислить вектор нормали, имея **U** и **V**, чтобы затем можно было вычислить векторное произведение? Поверьте мне, проще всего рассказать об этом с помощью матрицы. Итак, предположим, что вы хотите вычислить векторное произведение **U** и **V**. Для этого необходимо построить следующую матрицу:

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}.$$

Здесь **i**, **j** и **k** — единичные векторы, параллельные осям X, Y и Z соответственно.

Чтобы вычислить векторное произведение **U** и **V**, нужно просто найти определитель записанной выше матрицы:

$$\mathbf{N} = \left(u_y v_z - u_z v_y\right)\mathbf{i} + \left(u_z v_x - u_x v_z\right)\mathbf{j} + \left(u_x v_y - u_y v_x\right)\mathbf{k}.$$

Таким образом, **N** представляет собой линейную комбинацию трех скаляров, каждый их которых умножен на взаимно ортогональные единичные векторы, параллельные осям X, Y и Z соответственно. Поэтому результат векторного произведения можно записать в виде

$$\mathbf{N} = \left\langle u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x \right\rangle.$$

Нулевой вектор

Хотя вы, вероятно, не будете часто использовать нулевой вектор, все же рассмотрим его. *Нулевой вектор* имеет нулевую длину и не имеет направления. Это просто точка. Таким образом, в двухмерном пространстве нулевой вектор определяют как $\langle 0,0 \rangle$, в трехмерном — как $\langle 0,0,0 \rangle$ и т.д.

Радиус-вектор

Следующая тема, которую я хочу предложить вашему вниманию, — это радиус-вектор. Радиус-векторы полезно использовать при вычерчивании таких геометрических элементов, как линии, сегменты, кривые и т.д. Я использовал их при операциях отсечения и при вычислении пересечения сегментов в главе 13, "Основы физического моделирования". Взгляните на рис. B.11, где изображен радиус-вектор, который можно использовать для представления отрезка прямой.

Отрезок прямой идет из точки p_1 в точку p_2, **V** — вектор, идущий из точки p_1 в точку p_2, **v** — единичный вектор, идущий по направлению от точки p_1 в точку p_2. Теперь можно создать радиус-вектор **P**, который позволит нам начертить отрезок. Математически **P** записывается как $\mathbf{P} = \mathbf{P}_1 + \mathbf{v}t$. Здесь t — параметр, принимающий значения от 0 до $|\mathbf{V}|$. Если $t = 0$, $\mathbf{P} = \mathbf{P}_1 = \langle p_{1x}, p_{1y} \rangle$, т.е. указывает на начало сегмента. При $t = |\mathbf{V}|$ получаем $\mathbf{P} = \mathbf{P}_1 + |\mathbf{V}|\mathbf{v} = \langle p_{1x} + v_x, p_{1y} + v_y \rangle = \langle p_{2x}, p_{2y} \rangle$.

Векторы как линейные комбинации

Как уже отмечалось, при вычислении векторного произведения векторы можно записать в виде суммы: $\mathbf{U} = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$.

Здесь **i**, **j** и **k** — это единичные векторы, параллельные осям X, Y и Z соответственно (так называемые *орты*). В этой записи нет ничего магического, это просто другой способ записи векторов, которым можно при необходимости пользоваться. Все операции работают точно так же, как и при другом способе, например:

$$\mathbf{U} = 3\mathbf{i} + 2\mathbf{j} + 3\mathbf{k},$$
$$\mathbf{V} = -3\mathbf{i} - 5\mathbf{j} + 12\mathbf{k},$$
$$\mathbf{U} + \mathbf{V} = 0\mathbf{i} - 3\mathbf{j} + 15\mathbf{k} = \langle 0, -3, 15 \rangle.$$

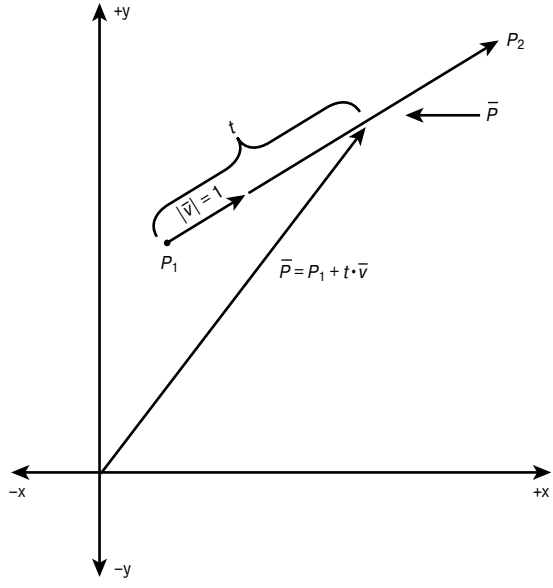


Рис. B.11. Радиус-векторы

Вы уже поняли, что это просто другой вид записи. Рассматривая вектор как линейную комбинацию независимых компонентов, помните, что, до тех пор пока каждый компонент имеет свой векторный коэффициент, компоненты нельзя “смешивать”. Таким образом, можно писать очень длинные выражения, а затем собирать общие члены и выносить за скобки векторы.

Вот и весь математический обзор!

ПРИЛОЖЕНИЕ Г

Азы C++

Язык программирования C++

C++ представляет собой усовершенствованный с помощью объектно-ориентированной технологии язык C. В каком-то смысле можно считать C++ расширенным вариантом C. Язык C++ имеет ряд усовершенствований (по сравнению с C).

- Классы.
- Наследование.
- Полиморфизм.

Давайте кратко рассмотрим каждое из этих свойств. *Классы* (classes) — это новый способ комбинации данных и функций. Обычно при программировании на C программист имеет дело со структурами данных, которые хранят данные, и функциями, оперирующими с ними, как показано на рис. Г.1,*а*. Однако в C++ и данные, и функции, оперирующие с данными, содержатся внутри единого класса, как показано на рис. Г.1,*б*. Почему это удобно? Потому что можно рассматривать экземпляр класса как *объект*, который имеет свойства и который может выполнять действия. Это более высокий уровень абстрагирования, чем используемый в C.

Второе важное свойство C++ — это *наследование*. Когда вы создаете классы, они дают абстрактную возможность создавать взаимосвязи между объектами класса и порождать один объект или класс из другого. В реальной жизни это приходится делать постоянно, почему бы это не сделать и в программировании? Например, вы могли бы иметь класс, называемый person, который будет содержать данные о человеке и иметь методы класса, чтобы оперировать с этими данными. Суть в том, что человек — понятие очень обобщенное. Мощь наследования вступает в игру, когда вы хотите создать два разных типа людей, например специалиста по программному обеспечению и специалиста в области аппаратных средств. Давайте назовем их sengineer (специалист по программному обеспечению) и hengineer (специалист в области аппаратных средств).

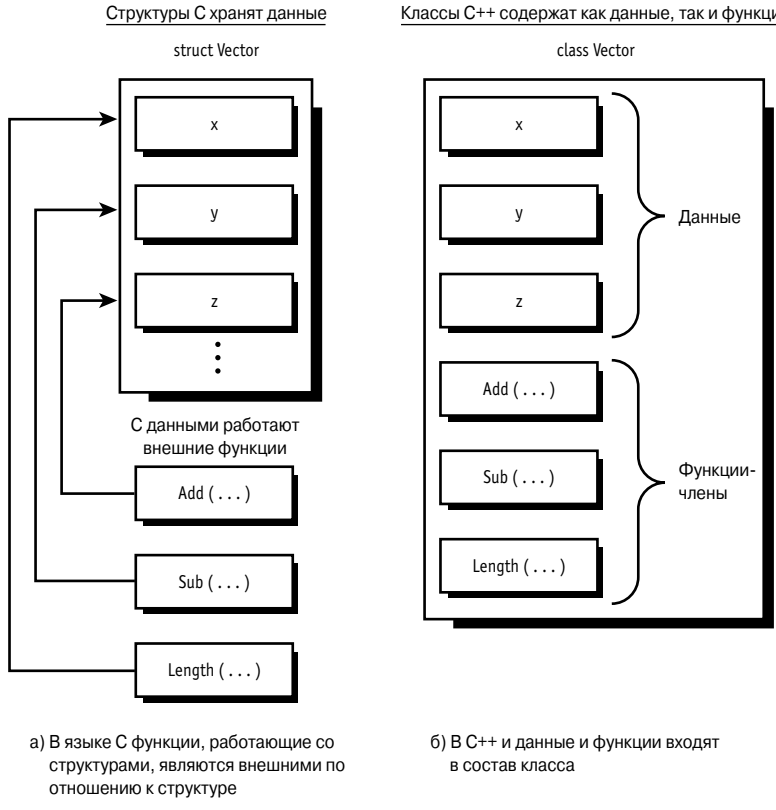


Рис. Г.1. Структура класса

На рис. Г.2 показана взаимосвязь между person, sengineer и hengineer.

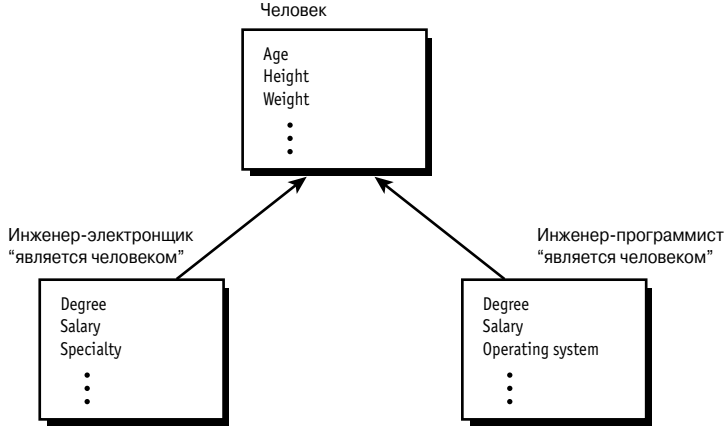


Рис. Г.2. Наследование класса

Посмотрим, каким образом два новых класса основаны на классе person. Оба специалиста, sengineer и hengineer, являются людьми, т.е. в основе лежит человек (person), но с дополнительными данными. Таким образом, вы наследуете все свойства person, но при

этом добавляете новые свойства, чтобы создать sengineer и hengineer. Это основа наследования. Можно построить более сложные объекты на основе существующих. Кроме того, существует *множественное наследование*, которое позволяет построить новый объект из нескольких подклассов.

И наконец, рассмотрим третье свойство C++ и объектно-ориентированного программирования — *полиморфизм*, что дословно означает "много форм". В контексте C++ полиморфизм подразумевает, что функции или операторы в зависимости от ситуации могут представлять собой разные вещи. Например, выражение (a+b) в C всегда означает, что нужно прибавить a к b. Также известно, что операция суммирования имеется в наличии только у ряда встроенных типов, таких, как int, float, short и т.п. В C вы не можете определить новый тип и применить к нему операцию сложения, а в C++ это вполне возможно. Следовательно, вы можете *перегрузить* операторы, такие, как +, –, *, /, [] и т.п., и заставить их делать разные вещи в зависимости от типа данных.

Кроме того, можно перегрузить функции. Например, вы записываете функцию Compute() следующим образом:

```
int Compute(float x, float y)
{
  // код
}
```

Функция принимает в качестве параметров два действительных числа; если передать ей целые числа, то они будут преобразованы в эквивалентные числа с плавающей точкой и лишь затем переданы функции. В C++ же можно сделать следующее определение:

```
int Compute (float x, float y)
{
  // код
}

int Compute (int x, int y)
{
  // код
}
```

Несмотря на то что эти функции имеют одно и то же имя, в качестве параметров они принимают данные разных типов. Исходя из этого, компилятор считает, что это совершенно разные функции, и поэтому передача функции целых чисел приводит к вызову второй функции, передача действительных чисел — к вызову первой. Если же вы вызываете функцию, передавая одновременно и целое и вещественное число, дело осложняется. При этом в игру вступают правила разрешения коллизий, которые позволяют компилятору решить, какую именно из двух функций вызвать.

Все это относится к C++. Конечно, существуют изменения в синтаксисе языка и множество новшеств, но чаще всего вам придется иметь дело с описанными концепциями.

Минимум, который необходимо знать о C++

C++ является чрезвычайно сложным языком, и потому слишком быстрое и излишнее использование новых технологий может привести к созданию ненадежных программ с разными видами утечек памяти, проблемами функционирования и т.д. Дело в том, что C++ — это язык "черных ящиков". Ряд процессов протекает за "сценой", и вы можете никогда не

обнаружить созданные вами ошибки. Однако если вы начнете использовать возможности C++ понемногу, не торопясь и по мере необходимости добавляя новые возможности в используемое вами подмножество языка, то останетесь вполне им довольны.

Единственная причина, по которой я написал это приложение по C++, состоит в том, что этот язык лежит в основе DirectX. Однако большая часть кода C++ инкапсулирована в интерфейсы COM, с которыми вы общаетесь посредством вызовов указателей функций, т.е. вызовов вида interface->function(). Если вы уже делали это, читая книгу, то этот странный синтаксис не должен удивлять вас. Если же вы никогда не встречались с указателями на функции, то глава 5, "Основы DirectX и COM", поможет вам разобраться в этом материале. Здесь же я хочу изложить только некоторые из многочисленных возможностей C++, чтобы вы лучше понимали его и имели достаточный объем знаний для начала работы с ним.

Я представлю вам несколько новых типов, обозначений, систему управления памятью, базовые классы, потоки ввода-вывода, перегрузку функций и операторов — и больше ничего. Поверьте, для начинающего этого более чем достаточно.

Новые типы, ключевые слова и обозначения

Давайте начнем с чего-нибудь простого, например с нового оператора комментария (//). Он стал частью стандарта C, поэтому вы его, скорее всего, уже использовали. Оператор // является оператором однострочного комментария в C++.

Комментарии

```
// это комментарий
```

Можно использовать и старый стиль комментария (/**/), если он вам больше нравится:

```
/* Многострочный комментарий языка C
   Весь этот текст является комментарием
 */
```

Константы

Для создания константы в обычном языке C можно выполнить одно из двух действий:

```
#define PI 3.14
```

или

```
float PI = 3.14
```

Проблема при использовании первого метода состоит в том, что PI не является действительной переменной с определенным типом. Это всего лишь символьное имя, которое препроцессор использует для подстановки в тексте, поэтому PI не имеет ни типа, ни размера и т.д. Проблема со вторым определением заключается в том, что эта переменная является записываемой (т.е. может изменяться в процессе работы программы без каких-либо предупреждений или сообщений об ошибке). C++ позволяет в такой ситуации использовать ключевое слово const, обеспечивающее создание переменной, которую можно только читать.

```
const float PI = 3.14;
```

Теперь вы можете использовать PI везде, где хотите; типом этой переменной является float, а ее размер определяется как sizeof(float). При этом вы не можете непосредственно изменить значение этой переменной. Это наилучший способ создания констант.

Ссылки

В языке C у вас часто возникает желание изменить значение переменной в функции, поэтому вы вынуждены передавать в функцию указатель на эту переменную.

```
int counter = 0;
void foo(int *x)
{
  (*x)++;
}
```

Если вызвать foo(&counter), то после вызова counter увеличится на 1. Итак, функция изменяет значение передаваемой ей переменной. В C++ для облегчения этого процесса существует новый тип переменной. Эту переменную называют *ссылкой* (reference) и обозначают в списке формальных параметров оператором получения адреса &.

```
int counter = 0;
void foo(int &x)
{
  x++;
}
```

Функция при этом вызывается очень просто:

```
foo(counter);
```

Обратите внимание, что больше ставить & перед counter не нужно. Здесь x становится альтернативным именем для любой передаваемой переменной. Следовательно, counter эквивалентен x и при вызове функции уже не требуется получение адреса.

Ссылки можно создавать и вне функций, например:

```
int x;
int &x_alias = x;
```

Здесь x_alias — это псевдоним x. Когда бы и где бы вы не использовали x, там же вы можете использовать и x_alias — это *одно и то же*.

Создание переменных “на лету”

Одним из новых свойств C++ является его способность создавать переменные внутри блоков программы, а не только на глобальном или функциональном уровне. Например, вот как можно написать цикл в C:

```
void Scan(void)
{
  int index;
  // ... Много строк кода ...

  // Наш цикл
  for (index = 0; index < 10; index++)
    Load_Data(index);

  // ... Здесь еще больше строк кода ...

} // Scan
```

В этой программе все корректно. Однако переменная index, объявленная в самом начале кода, используется в качестве индекса цикла в одном-единственном месте програм-

мы, достаточно далеко от своего объявления. Разработчики C++ увидели недостатки такой записи и разрешили определять переменные поближе к тому месту, где их используют. Кроме того, переменная, которую используют в одном из блоков программы, невидима для других блоков. Например, у нас есть такой набор блоков программы:

```
void Scope(void)
{
int x = 1, y = 2; // Глобальная область видимости
printf("\nBefore Block A: Global Scope x=%d, y=%d", x, y);
   { // Блок A
      int x = 3, y = 4;
      printf("\nIn Block A: x=%d, y=%d",x,y);
   } // Конец блока A
printf("\nAfter Block A:Global Scope x=%d, y=%d", x, y);
   { // Блок B
      int x = 5, y = 6;
      printf("\nIn Block B: x=%d, y=%d", x, y);
   } // Конец блока B
printf("\nAfter Block B: Global Scope x=%d, y=%d", x, y);
}  // Scope
```

В памяти переменные x, y хранятся в трех различных местах, т.е. существуют в трех версиях. Первые x и y определены глобально. Однако после входа в блок A они вытесняются из области видимости переменными x и y из локальной области видимости. Затем, после выхода из блока A, в область видимости возвращаются глобальные x и y, которые тут же выходят из нее, как только начинается блок B. Применяя области видимости блоков, вы можете более корректно локализовать переменные и их использование. При этом вам не нужно заботиться о новых именах переменных — можно продолжать использовать x, y или что-либо еще, не беспокоясь, что новые переменные испортят глобальные переменные с теми же именами.

Одна из самых приятных вещей в новых правилах работы с областями видимости состоит в том, что переменную можно создавать оперативно, “на лету”. Например, вот тот же цикл с переменной цикла index, что и ранее, но реализованный на C++:

```
for (int index = 0; index < 10; index++)
   Load_Data(index);
```

Разве это не замечательно? Я определил index именно там, где я его использую, а не в начале функции, как раньше. Только не слишком увлекайтесь, используя эту возможность!

Управление памятью

Язык C++ имеет новую систему управления памятью, основанную на операторах new и delete. В основном они эквивалентны функциям malloc() и free(), но гораздо интеллектуальнее их, поскольку учитывают тип создаваемых или удаляемых данных. Приведу пару примеров.

В языке C, чтобы выделить память для 1000 переменных типа int из кучи, мы используем функцию malloc().

```
int *x = (int*)malloc(1000*sizeof(int));
```

А вот как то же самое делается в C++:

```
int *x = new int[1000];
```

Это уже намного лучше! Оператор new знает, что должен вернуть указатель на int (int*), так что вам не нужно выполнять приведение типов самому. Далее, чтобы освободить память в C, вы должны сделать следующее:

```
free(x);
```

В C++ это делается так:

```
delete[] x;
```

В сущности, это то же самое, но выполняется новым оператором. В программе вы можете использовать средства любого языка — как C, так и C++. Но не смешивайте их! Не освобождайте память, выделенную посредством new, при помощи вызова free(), как не освобождайте память, выделенную malloc(), с помощью delete.

Потоки ввода-вывода

Мне нравится функция printf(). Ну что может быть проще?

```
printf("Hello, world!\n");
```

Единственная проблема при использовании printf() состоит в том, чтобы запомнить и правильно использовать все эти спецификаторы формата — %d, %x, %u и т.д. Их слишком много. Но еще хуже scanf(), потому что при ее применении легко забыть о необходимости использовать адрес переменной для хранения получаемых данных.

```
int x;
scanf("%d", x);
```

Так поступать нельзя! Необходимо использовать адрес переменной x, поэтому правильная запись будет выглядеть так:

```
scanf("%d", &x);
```

Я уверен, что вы неоднократно допускали эту ошибку. Адрес оператора не используют только при работе со строками, потому что в этом случае имя переменной и есть адрес. Если вспомнить об этом, становится понятно, почему в C++ был создан новый класс потоков ввода-вывода. Этот класс сам может разобраться, с какими типами переменных он работает, так что нет необходимости сообщать ему об этом. Классы потоков ввода-вывода определены в заголовочном файле iostream.h, поэтому при работе с потоками не забудьте включить его в вашу программу на языке C++. Как только это сделано, программист получает доступ к стандартным потокам cin, count, cerr и cprn, показанным в табл. Г.1.

Таблица Г.1. Потоки I/O C++

Имя потока	*Механизм*	*Имя в C*	*Назначение*
cin	Клавиатура	stdin	Стандартный ввод
cout	Экран	stdout	Стандартный вывод
cerr	Экран	stderr	Стандартный вывод сообщений об ошибках
cprn	Принтер	stdprn	Принтер

Потоки ввода-вывода используют перегруженные операторы << и >>. В языке C эти операторы обычно означают побитовый сдвиг чисел влево и вправо, но в контексте потоков ввода-вывода их используют для отправки и получения данных. Ниже приведены примеры использования стандартного вывода.

```
int  i;
float f;
char  c;
char  string[80];

// В C
printf ("\nHello world!");
// В C++
cout << "\nHello world!";

// В C
printf ("%d", i);
// В C++
cout << i;

// В C
printf ("%d, %f, %c, %s", i, f, c, string);
// В C++
cout << i << ", " << f << ", " << c << ", " << string;
```

Разве это не замечательно? Вам не нужен никакой спецификатор типа, потому что cout сам знает тип данных и выполняет эту работу за вас. Единственной странностью в синтаксисе работы с потоками является то, что в C++ вы можете использовать сколько угодно операторов << в одном выражении. Дело в том, что каждая операция возвращает сам поток, так что добавлять очередные операторы << можно бесконечно долго. Следует заметить, что при использовании потоков ввода-вывода вы должны сами заботиться о разделении переменных сроками типа ",". При желании можно отформатировать код таким образом, что каждый оператор << будет располагаться на отдельной строке.

```
cout <<  i
   << ","
   << f
   << ","
   << c
   << ","
   << string;
```

Поток ввода работает в основном так же, но при этом используется оператор >>. Ниже приведено несколько примеров использования потока ввода.

```
int  i;
float f;
char  c;
char  string[80];

// В C
printf ("\nWhat is your age?");
scanf ("%d", &i);
// В C++
cout << "\nWhat is your age?";
cin  >> i;

// В C
printf ("\nWhat is your name and grade?");
```

```
scanf ("%s %c", string, &c);
// В C++
cout << "\nWhat is your name and grade??";
cin  >> string >> c;
```

Проще, чем в C, не правда ли? Конечно, это только маленькая демонстрация; на самом деле потоки ввода-вывода умеют множество разных вещей, так что лучше познакомьтесь с ними отдельно — это стоит затраченного времени.

Классы

Классы являются самым важным дополнением в C++, и именно на них базируется объектно-ориентированное программирование. Ранее я говорил, что *класс* — это просто контейнер для хранения данных и методов (часто называемых *функциями-членами*), которые оперируют с этими данными.

Структуры в C++

Давайте начнем изучать классы со стандартных структур, постепенно усложняя их. В C структуру определяют примерно так:

```
struct Point
{
  int x, y;
};
```

После этого можно создать экземпляр структуры:

```
struct Point p1;
```

Эта операция создает экземпляр, или объект, структуры Point и дает ему имя p1. В C++ ключевое слово struct для создания экземпляра использовать не нужно, так что создать объект можно просто как

```
Point p1;
```

Причина такого упрощения кроется в том, что в C++, по сути, создан новый тип Point, так что теперь больше не нужно пояснять, что это структура. Того же результата в C можно попытаться добиться, например, так:

```
typedef struct tagPOINT
{
  int x, y;
} Point;

Point p1;
```

Классы аналогичны структурам в том, что их определения являются определением типов.

Простой класс

Класс в C++ определяют при помощи ключевого слова class, например:

```
class Point
{
public:
  int x, y;
};

Point p1;
```

Эта запись, по сути, идентична записи структуры Point; фактически обе версии p1 работают одинаково. Например, для получения доступа к данным используют обычный синтаксис:

```
p1.x = 5;
p1.y = 6;
```

И, конечно же, аналогично работают указатели. Поэтому вот как определяется указатель на объект типа Point и создается соответствующий динамический объект:

```
Point *p1;
p1 = new Point;
```

А так осуществляется присвоение значений полям x и y:

```
p1->x = 5;
p1->y = 6;
```

Отсюда следует, что классы и структуры идентичны в контексте доступа к элементам открытых данных. Ключевой термин — *открытый* (public). Что же он означает? Как вы могли заметить в предыдущем примере, класс Point определен следующим образом:

```
class Point
{
public:
  int x, y;
};
```

Здесь главное заключается в ключевом слове public, которое стоит в самом верху, перед всеми объявлениями. Оно определяет видимость переменных и функций-членов. Существует несколько спецификаторов видимости, но обычно используют только два — public и private.

Открытые и закрытые члены класса

Если вы поместили ключевое слово public в начале определения класса и размещаете в этом классе только данные, то вы получите стандартную структуру. Таким образом, структуры являются классами с общедоступными членами. Это означает, что обращаться к членам класса может любой код, а данные никак не скрыты и не инкапсулированы. Закрытые же члены данных класса доступны для обращения только функциям, являющимся членами класса. Например, рассмотрим такой класс:

```
class Vector3D
{
public:
  int x, y, z; // Кто угодно может изменять эти данные

private:
    int reference_count; // Эти данные скрыты
};
```

Класс Vector3D состоит из двух частей: области открытых и закрытых данных. Область открытых данных имеет три поля, которые можно изменять: x, y, z. В области закрытых данных в наличии одно скрытое поле reference_count. Это поле скрыто для всех, за исключением функций-членов класса. Таким образом, если бы вы написали код

```
Vector3D v;

v. reference_count = 1; // Недопустимо!
```

то компилятор сообщил бы об ошибке. В таком случае, чем же хороши закрытые переменные, если к ним нет доступа? Именно этим они и хороши. Они позволяют разрабатывать класс как "черный ящик", к внутренностям которого никто не может добраться и несанкционированно изменить его состояние. Чтобы получить доступ к закрытым членам класса, необходимо добавить в класс функции-члены, или методы.

Функции-члены класса

Функция-член, или *метод*, — это определенная в классе функция, которая работает с данными этого класса, например:

```
class Vector3D
{
public:
  int x, y, z; // Открытые данные

  // Функция-член
  int length(void)
  {
    return (sqrt(x*x + y*y + z*z);
  } // length
private:
  int reference_count; // Закрытые данные
};
```

Обратите внимание на выделенную функцию-член length(), которую я определил прямо в классе. Давайте посмотрим, как ее можно использовать.

```
vector3D v; // Создать вектор

// Присвоить значения
v.x = 1;
v.y = 2;
v.z = 3;

// Это самая интересная часть
printf ("\nlength = %d", v.length());
```

Вы вызываете функцию-член так же, как обращаетесь к элементу. Если же v является указателем, вызов осуществляется следующим образом:

```
v->length();
```

Вы можете сказать, что у вас около 100 функций, которые должны иметь доступ к данным класса, и вы просто не в состоянии все их разместить в классе. Думаю, если бы вы захотели, то смогли бы, но я согласен, что так очень легко запутаться. Однако функции-члены класса можно определить и вне определения класса. Вскоре мы именно так и сделаем. А пока я хочу добавить другую функцию-член, чтобы показать, как можно получить доступ к закрытому члену класса reference_count.

```
class Vector3D
{
public:
  int x, y, z; // Открытые данные

  // Функция-член
```

```
  int length(void)
  {
    return (sqrt(x*x + y*y + z*z);
  } // length

  void addref (void)
  {
    // Эта функция увеличивает счетчик ссылок
    reference_count++;
  } // addref
private:
  int reference_count; // Закрытые данные
};
```

Вы обращаетесь к reference_count посредством функции-члена addref(). Может быть, этот подход покажется вам излишне сложным, но поверьте, это очень удобно. Теперь пользователь вашего класса не может сделать какую-то глупость с вашим объектом. Обращение к закрытым членам класса всегда идет через ваши функции доступа.

Вызывающая программа не может изменить счетчик ссылок каким-то иным способом, потому что reference_count — закрытый член класса. Доступ к нему могут иметь только функции-члены класса.

Теперь, я думаю, вы понимаете всю мощь классов. Вы можете наполнить их данными, подобно структуре, добавить внутрь классов функции, которые оперируют с этими данными, и спрятать данные от доступа извне. Разве это не замечательно?

Конструкторы и деструкторы

Если вы программируете на C больше недели, то я больше чем уверен, что вы сотни раз делали одну и ту же операцию — инициализировали структуру. Например, допустим, что вы создаете структуру Person:

```
struct Person
{
  int  age;
  char *address;
  int  salary;
};

Person people[1000];
```

Теперь вам необходимо инициализировать 1000 структур people. Может быть, вы будете делать это так:

```
for (int index = 0; index < 1000; index++)
{
  people [index].age    = 18;
  people [index].address = NULL;
  people [index].salary  = 35000;

} // for index
```

Что произойдет, если вы забыли инициализировать данные и приступили к использованию структур? Скорее всего, к вам придет старый дядюшка General Protection Fault. А что, если вы распределили память для поля адреса объекта

```
people[20].address = malloc(1000);
```

и потом, забыв об этом, выделили память еще раз:

```
people[20].address = malloc(4000);
```

В итоге потеряны тысячи байт памяти. В этой ситуации перед выделением памяти нужно было освободить старую память посредством вызова функции free():

```
free(people[20].address);
```

Я думаю, вы не раз совершали такие ошибки. C++ разрешает данную проблему предоставлением двух функций-членов, которые автоматически вызываются при создании и уничтожении объекта класса. Это *конструкторы* и *деструкторы* классов.

Конструктор вызывается при создании объекта класса. Например, при выполнении кода

```
Vector3D v;
```

вызывается конструктор по умолчанию, который в данном случае ничего не делает. Аналогично, когда v выходит из поля видимости (например, если функция, определившая локальный объект v, завершает работу или если v является глобальным объектом, когда прекращается работа программы), вызывается деструктор по умолчанию, который тоже ничего не делает. Чтобы увидеть конструктор и деструктор в действии, нужно их написать. Если вы не хотите этого делать — не делайте. При желании вы можете определить как один конструктор, так и один деструктор, а также обе функции.

Создание конструктора

Воспользуемся структурой person, преобразованной в класс.

```
class Person
{
public:
  int  age;
  char *address;
  int  salary;

  // Конструктор по умолчанию
  // Конструкторы могут содержать как пустой список
  // параметров, так и любой набор параметров
  // конструкторы никогда не возвращают никакого
  // значения, даже void
  Person()
  {
    age    = 0;
    address = NULL
    salary = 35000;
  } // Person
}
```

Обратите внимание, что конструктор имеет то же имя, что и класс, в данном случае Person. Это не совпадение, это — правило! Обратите также внимание на то, что конструктор ничего не возвращает. Так и должно быть. Однако конструктор может получать параметры. В данном случае у нас нет параметров, но могут быть и конструкторы с параметрами. Фактически можно иметь неограниченное число различных конструкторов, каждый со своим списком параметров. В любом случае, чтобы создать и автоматически инициализировать объект, необходимо его объявить:

```
Person person1;
```

Конструктор при этом вызовется автоматически:

```
person1.age     = 0;
person1. Address = NULL;
person1.salary   = 35000;
```

При создании 1000 объектов

```
Person people[1000];
```

конструктор будет вызван для каждого отдельного экземпляра Person, и все 1000 объектов будут инициализированы без какого-либо участия с вашей стороны.

Теперь давайте вспомним, что я рассказывал о перегружаемых функциях. Конструкторы тоже можно перегружать. Следовательно, если вы хотите иметь конструктор, чтобы объекту во время его создания можно было передать такие параметры, как возраст, адрес и оклад, вам необходимо сделать следующее:

```
class Person
{
public:
  int  age;
  char *address;
  int  salary;

  // Конструктор по умолчанию
  Person()
  {
    age    = 0;
    address = NULL
    salary = 35000;
  } // Person()

  // Новый, более мощный конструктор
  Person(int new_age, char *new_address, int new_salary)
  {
    age = new_age;
    // Выделим память для адреса и присвоим значение
    address = new char[strlen(new_address)+1];
    strcpy(address, new_address);
    salary = new_salary;
  } // Person(int,char*,int)
}
```

Теперь у вас есть два конструктора, один из которых не получает ни одного параметра, а другой — сразу три. Ниже приведен пример создания экземпляра Person для человека, которому 24 года, который живет по адресу 500 Maple Street и зарабатывает $52000 в год.

```
Person person2(24,"500 Maple Street",52000)
```

Очень элегантно, не так ли? Конечно, можно подумать, что инициализировать структуры C следует с использованием другого синтаксиса:

```
Person person = {24, "500 Maple Street", 52000}
```

Но как тогда решить вопросы с распределением памяти? С копированием строк? Обычный C может делать побитовую копию и не более, а C++ дает вам возможность запускать код при создании объекта. Благодаря этому появляется значительно больше возможностей для управления объектами.

Создание деструктора

После того как объект создан, рано или поздно он будет уничтожен. В C обычно для этого создается функция, которая освобождает все захваченные ресурсы и уничтожает объект, но в C++ объект уничтожает сам себя путем вызова деструктора. Написать деструктор еще проще, чем конструктор, поскольку здесь меньше вариантов — деструкторы имеют только один вид:

```
~classname();
```

У деструкторов нет ни параметров, ни возвращаемого значения. Исключений нет! Памятуя об этом, давайте добавим деструктор в наш класс Person:

```
class Person
{
public:
  int  age;
  char *address;
  int  salary;

  // Конструктор по умолчанию
  Person()
  {
    age    = 0;
    address = NULL
    salary = 35000;
  } // Person()

  // Новый, более мощный конструктор
  Person(int new_age, char *new_address, int new_salary)
  {
    age = new_age;
    // Выделим память для адреса и присвоим значение
    address = new char[strlen(new_address)+1];
    strcpy(address, new_address);
    salary = new_salary;
  } // Person(int,char*,int)

  // Деструктор
  ~Person()
  {
    delete[] address;
  } // ~Person

}
```

Следует отметить, что нет никаких оговорок относительно кода внутри деструктора, т.е. вы можете делать все, что хотите. С новым деструктором, например, можно не беспокоиться об освобождении памяти. Так, в языке C, если в некоторой функции вы создаете структуру с внутренними указателями, которым выделяется память, а затем выходите из функции без освобождения этой выделенной структуре памяти, вы получаете *утечку памяти*, как в приведенном ниже фрагменте кода.

```
struct {
  char *name;
```

```
  char *ext;
  } filename;

void foo()
{
  filename file; // Имя файла

  file.name = malloc(80);
  file.ext  = malloc(4);

} // foo
```

Структура file уничтожается, и выделенные 84 байта оказываются навсегда потерянными. Но в C++ при наличии деструктора это невозможно. Компилятор гарантирует вызов деструктора, который освобождает память.

Я познакомил вас с основной информацией по конструкторам и деструкторам. На самом деле материал по этой теме гораздо обширнее. Существуют, например, специальные конструкторы — скажем, конструкторы копирования и т.д. Но и того, что вы уже знаете, достаточно, чтобы начать работу. Что касается деструкторов, то существует только один тип, который я вам и продемонстрировал.

Оператор разрешения области видимости

В языке C++ имеется новый оператор, именуемый *оператором разрешения области видимости*, который обозначается двойным двоеточием (::). Он используется для обращения к функциям класса и членам данных в области видимости класса. Не ломайте голову над тем, что это означает. Я просто покажу вам, как его использовать для определения функций класса вне класса.

До сих пор вы определяли функции-члены класса прямо внутри определения класса. Хотя это вполне приемлемо для небольших классов, для больших классов это весьма проблематично. Поэтому вы можете определять функции-члены класса вне класса, только при этом нужно сообщить компилятору, что это функции класса, а не обычные функции уровня файла. Это делается с помощью оператора разрешения области видимости и следующего синтаксиса:

```
return_type class_name::function_name(parm_list)
{
  // Тело функции
}
```

Конечно, в самом классе вы должны еще объявить функцию с соответствующим прототипом (без оператора разрешения области видимости и имени класса), но тело функции при этом находится в другом месте. Давайте посмотрим, как это делается, на примере нашего класса Person. Ниже приведен класс с удаленными из него телами функций.

```
class Person
{
public:
  int  age;
  char *address;
  int  salary;

  Person();
```

```
  Person(int new_age, char *new_address, int new_salary);
  ~Person();
}
```

А вот тела функций, которые размещаются со всеми другими функциями после определения класса:

```
// Конструктор по умолчанию
Person::Person()
{
  age    = 0;
  address = NULL
  salary = 35000;
} // Person()

// Новый, более мощный конструктор
Person::Person(int new_age, char *new_address, int new_salary)
{
  age = new_age;
  // Выделим память для адреса и присвоим значение
  address = new char[strlen(new_address)+1];
  strcpy(address, new_address);
  salary = new_salary;
} // Person(int,char*,int)

// Деструктор
Person::~Person()
{
  delete[] address;
} // ~Person
```

СЕКРЕТ

Большинство программистов ставят перед именем класса заглавную букву C. Обычно я делаю то же самое, но не хотел заострять на этом внимание. Поэтому в процессе реального программирования я, вероятно, назову класс CPerson вместо Person. Или, может быть, CPERSON, т.е. воспользуюсь заглавными буквами.

Перегрузка функций и операторов

Последней темой, о которой я хотел бы поговорить, является *перегрузка*, которая выступает в двух видах: *перегрузка функций* и *перегрузка операторов*. Рамки книги не позволяют подробно объяснять перегрузку операторов, поэтому я приведу общий пример. Давайте представим, что у вас есть класс Vector3D, вы хотите сложить два вектора, v1+v2, и сохранить сумму как v3. Вы должны сделать нечто наподобие следующего кода:

```
Vector3D v1 = {1,2,5},
       v2 = {5,9,8},
       v3 = {0,0,0};

// Определяем дополнительную функцию,
// которая может быть функцией класса
Vector3D Vector3D_Add(Vector3D v1, Vector3D v2)
{
  Vector3D sum; // используется для хранения суммы
```

```
  sum.x = v1.x+v2.x;
  sum.y = v1.y+v2.y;
  sum.z = v1.z+v2.z;

  return (sum);

} // Vector3D_Add
```

Тогда для того, чтобы сложить векторы при помощи функции, вы должны написать следующий код:

```
v3 = Vector3D_Add(v1, v2);
```

Это грубый, но работоспособный метод. Однако в C++ оператор + можно перегрузить и создать его версию для сложения векторов, т.е. векторы можно будет складывать так же, как и обычные числа:

```
v3 = v1+v2;
```

Далее приведен синтаксис перегруженной операторной функции, но, чтобы получить об этом более полное представление, рекомендую прочитать книгу по языку C++.

```
class Vector3D
{
public:
  int x, y, z;

  // Функция-член
  int length(void) {return sqrt(x*x + y*y + z*z);}

  // перегруженный оператор +
  Vector3D operator+(Vector3D& v2)
  {
    Vector3D sum; // используется для хранения суммы
    sum.x = x+v2.x;
    sum.y = y+v2.y;
    sum.z = z+v2.z;
    return sum;
  }

private:
   int reference_count; // Скрытые данные

};
```

Заметьте, что неявный первый параметр функции-члена (включая операторы) является самим вызывающим объектом, поэтому список параметров содержит только v2. Перегрузка операторов — очень мощное средство. Благодаря ей можно реально создавать новые типы данных и операторы, что позволяет выполнять различные виды операций без вызова функций.

Перегрузку функции вы уже видели, когда обсуждались конструкторы. Перегрузка функции — это не что иное, как использование одного и того же имени для написания двух или более функций, которые различаются своими аргументами. Предположим, вы хотите написать функцию Plot_Pixel(), которая имеет следующие функциональные возможности: если вы вызываете ее без параметров, она просто выводит пиксель в текущей позиции курсора, но если передать ей координаты x, y, то она выводит пиксель в соответствующей позиции. Ниже показано, как это следует кодировать.

```
int cursor_x, cursor_y; // Положение глобального курсора

//первая версия Plot_Pixel
void Plot_Pixel(void)
{
  // Выводим пиксель в позиции курсора
  plot(cursor_x, cursor_y);
}

// вторая версия Plot_Pixel
void Plot_Pixel(int x, int y)
{
  // Выводим пиксель в переданной точке и
  // соответственно перемещаем курсор
  plot(cursor_x=x,cursor_y=y);
}
```

После такого определения вы можете вызвать функцию Plot_Pixel() одним из следующих способов.

```
Plot_Pixel(10,10); // Вызов версии 2
Plot_Pixel();      // Вызов версии 1
```

СЕКРЕТ

Компилятор различает функции с одинаковыми именами, поскольку *реальное* имя функции состоит не только из имени функции, но и учитывает число и тип ее аргументов, что и создает уникальное имя (сигнатуру) в пространстве имен компилятора.

Резюме

Вот мы и завершили наше небольшое знакомство с языком программирования C++. Если бы Роберт Лафор (Robert Lafore) — автор одной из самых популярных книг по языку C++ — прочитал это приложение, он, вероятно, убил бы меня за слишком вольное обращение с языком, но зато вы получили некоторое представление о языке C++ и теперь, по крайней мере, сможете разобраться в коде C++, если даже пока и не сможете написать его сами.

ПРИЛОЖЕНИЕ Д

Ресурсы по программированию игр

Ниже приведен набор ресурсов, которые могут оказаться полезными при программировании игр.

Web-узлы по программированию игр

Существует сотни крупных узлов, и привести их все в одной книге невозможно. Вот самые любимые мною.

GameDev.Net	http://www.gamedev.net/
The Official MAME Page	http://www.mame.net/
The Games Domain	http://www.gamesdomain.com/
The Coding Nexus	http://www.gamesdomain.com/gamedev/gprog.html
Конференция разработчиков компьютерных игр	http://www.gdconf.com
Конференция разработчиков игр компании Xtreme	http://www.xgdc.com

Откуда можно загрузить информацию

Программист игр должен иметь доступ к новым играм, инструментальным средствам, утилитам и другим материалам. Ниже приведен список узлов, откуда я предпочитаю загружать то, что мне нужно.

eGameZone	http://www.egamezone.net
Happy Puppy	http://www.happypuppy.c om
Game Pen	http://www.gamepen.com/topten.asp
Ziff Davis Net	http://www.zdnet.com/swlib/games.html
Adrenaline Vault	http://www.avault.com/pcrl/
Download.Com	http://www.download.com/pc/cdoor/0,323,0-17,00.html?st.dl.fd.cats.cat17
Jumbo.Com	http://www.jumbo.com/games/g2/
GT Interactive	http://www.gtgames.com
Epic Megagames	http://www.epicgames.com
Cnet	http://www.cnet.com
WinFiles.com	http://www.winfiles.com
eGames	http://www.egames.com

2D/3D-процессоры

В Web есть только одно место, где сделан акцент на разработке трехмерных игровых процессоров. Это *The 3D Engine List*, который содержит ссылки на технологии различного уровня. И что самое поразительное — большинство авторов дают разрешение использовать свои разработки бесплатно! Вот адрес этого узла: http://cg.cs.tu-berlin.de/~ki/engines.html.

Ниже приведены ссылки на некоторые специфические игровые 2D/3D-процессоры.

Genesis 3D Engine	http://www.genesis3d.com
SciTech MGL	http://www.scitechsoft.com
Crystal Space	http://crystal.linuxgames.com/

Книги по программированию игр

Существует множество книг по графике, звуку, мультимедиа и разработке игр, но покупать их все очень дорого. Поэтому ниже даны адреса некоторых узлов, где содержатся обзоры таких книг и с которых можно загрузить книги по программированию игр.

Games Domain Bookstore	http://www.gamesdomain.com/gameev/gdevbook.html
Premier Publishing Game Development Series	http://www.premierpressbooks.com/gamedevseries.asp

Microsoft DirectX

Несомненно, самый крупный Web-узел в мире у компании Microsoft. На нем находятся тысячи страниц, разделов, узлов FTP и т.д. Однако главной страницей, которая должна вас интересовать, является DirectX Multimedia Expo, которую можно найти по адресу: http://www.microsoft.com/directx/.

На этой странице вы найдете самые последние новости и сможете загрузить самые последние версии DirectX, DirectMedia и исправления для предыдущих версий. Думаю, вам будет несложно каждую неделю посвящать один час просмотру этой информации — во всяком случае, вреда от этого не будет. Это поможет вам быть в курсе всех новинок DirectX. И, конечно же, не забывайте о новом узле Xbox по адресу: http://www.xbox.com/.

Конференции Usenet

Я никогда долго не участвую в телеконференциях Internet, поскольку это слишком медленное средство коммуникации (почти такое же медленное, как и чтение обычных текстовых материалов). Но ниже все же приведено несколько названий телеконференций, на которые стоит обратить внимание.

alt.games
rec.games.programmer
comp.graphics.algorithms
comp.graphics.animation
comp.ai.games

Если вы никогда не читали материалы таких сетевых конференций, почитайте. Для этого вам понадобится программа для чтения групп новостей, которая может загружать информацию и позволит вам читать сообщения по выбранной теме. Большинство Web-броузеров, таких, как Netscape Navigator и Internet Explorer, имеют встроенные программы чтения групп новостей. Просто обратитесь к справочной системе этих программ и выясните, как настроить ваш броузер для чтения конференций.

Blues News

Примерно 99,9% содержимого Internet — это хлам. WWW использует огромное количество людей, главным образом для того, чтобы просто поболтать и поразвлечься. Но в Internet есть несколько узлов, зайдя на которые, вы не потратите время попусту. Один из них — *Blues News*, на котором размещены пиктограммы компаний, работающих в различных отраслях, и ежедневный обзор новостей. Загляните на этот узел по адресу: http://www.bluesnew.com.

Журналы, посвященные разработке игр

Насколько мне известно, на английском языке есть только два журнала, посвященных разработке игр. Первым и наиболее крупным является журнал *Game Developer*, который выходит ежемесячно и содержит статьи по программированию игр, трехмерному моделированию и т.д. Его Web-узел находится по адресу: http://www.gdmag.com.

Ради развлечения можно посетить узел *Gamasutra* (книга по сексу программистов игр) по адресу: http://www.gamasutra.com.

Разработчики игровых Web-узлов

И последнее, о чем стоит подумать при создании игры, — это о ее Web-узле. Если вы пытаетесь продавать свою игру самостоятельно как условно-бесплатную, то наличие мини-узла, на котором демонстрируется игра, будет очень важным подспорьем. Вы можете сами знать, как использовать FrontPage или простой Web-редактор в Netscape, но если хотите иметь действительно привлекательный Web-узел для демонстрации вашей игры, то необходимо сделать это профессионально. Я видел много действительно прекрасных игр, которые были ужасно представлены в Web.

Компания, услугами которой пользуюсь лично я, — *Belm Design Group*. Она может помочь вам создать узел для вашей игры; обычно это стоит от 500 до 3000 долларов. Вот ее адрес: http://www.belmdesigngroup.com.

Xtreme Games LLC

Название моей компании Xtreme Games LLC. Мы разрабатываем 2D/3D-игры на платформе PC. Вы можете найти наш Web-узел по адресу: http://www.xgames3d.com.

Здесь вы найдете статьи по трехмерной графике, искусственному интеллекту, физике, DirectX и многому другому. Здесь же я размещаю все изменения или дополнения к книге, которую вы держите в руках.

Компания Xtreme Games LLC создает и продает игры. Поэтому, если вы считаете, что у вас есть хорошая игра, то обратитесь с информацией об авторской разработке игр в Xtreme. *Мы также оказываем техническую помощь разработчикам*.

Я создал две новые компании для предоставления помощи разработчикам игр. Компания eGameZone.net задумана как место оперативного распространения игр и находится по адресу: http://www.egamezone.net. Другая компания, NuRvE Networks, занимается созданием новых поколений популярных во всем мире игр и сетевых игр. Ее адрес: http://www.nurve.net.

И наконец, напоминаю еще раз адрес моей электронной почты: ceo@xgames3d.com.

Предметный указатель

3

3D, *30*
3D Studio Max, *42*

A

Adobe Illustrator, *42*
Adobe Photoshop, *42*

B

BMP, *313*
  Заголовок, *313*
  Загрузка файла, *313*

C

C++, *845–63*
  Деструктор, *859*
  Класс, *853*
  Комментарии, *848*
  Константы, *848*
  Конструктор, *857*
  Множественное наследование, *847*
  Наследование, *845*
  Оператор разрешения области видимости, *860*
  Перегрузка, *861*
  Полиморфизм, *847*
  Ссылка, *849*
  Управление памятью, *850*
Caligari TrueSpace, *42*
COM, *200–211*
const, *69*
Corel Photo-Paint, *42*

D

DCOM, *218*
Direct3DIM, *199*
Direct3DRM, *199*
DirectDraw, *198*; *219–56*; *257–364*
  Видеорежим, *231*
  Задний буфер, *220*
  Палитра, *235*
  Поверхность, *220*; *239*
  Создание объекта, *223*
DirectInput, *199*; *501–34*
  Джойстик, *520–34*
  Клавиатура, *506–14*
  Мышь, *515–19*
  Опрос устройства, *504*
  Режимы получения данных, *505*
DirectMusic, *199*; *571–75*
DirectPlay, *199*
DirectShow, *200*
DirectSound, *198*; *552–56*
DirectSound3D, *198*
DirectX, *30*; *195*
  Совместное использование с GDI, *350*
DirectX Audio, *200*
DirectX Graphics, *200*
DLS, *542*

G

GDI, *81*; *126*; *157–91*
  Совместное использование с DirectX, *350*
GUID, *204*

H

HAL, *198*
HEL, *198*

M

MIDI, *43*; *542*; *548*; *575–80*

P

Paint Shop Pro, *42*
ProCreate Painter, *42*

R

RGB-форматы, *258*

S

SMP, *624*
Sound Forge Xp, *42*

W

WAV, *566*
WinProc, *86*

Б

Билинейная фильтрация, *338*
Блиттинг, *240*; *286–99*
Брезенхама алгоритм, *367*

В

Вектор, *836*
  Векторное произведение, *840*
  Длина, *837*
  Норма, *837*
  Орт, *842*
  Радиус-вектор, *842*
  Скалярное произведение, *839*
  Сложение, *838*
Венгерская нотация, *68*
Взаимоисключение, *645*
Видеопамять, *131*
Видеоповерхность, *239*
Вращение, *334*

Г

Гамма-коррекция, *348*
Генетические алгоритмы, *697*
Глобальные переменные, *39*
Глубина цвета, *131*
Графический контекст, *90*

Д

Двойная буферизация, *240*; *271*
Двухсвязный список, *599*
Дерево, *603*
  Бинарное, *603*
  Порядок обхода, *609*
Дескриптор, *80*
Дескриптор контекста устройства, *158*
Джойстик, *520–34*
  Мертвая зона, *529*
Диалог, *100*

З

Задний буфер, *276*
Заполнение многоугольника, *412*
Звук, *100*; *110*; *541*
  Амплитуда, *543*
  Волноводный синтез, *550*
  Запись, *551*
  Синтезированный, *547*
  Скорость, *543*
  Табличный синтез, *549*
  Цифровой, *546*
  Частота, *543*
  Частота выборки, *546*

И

Игра
  Asteroids, *32*; *648*
  Civilization, *32*
  Command and Conquer, *677*
  Core Wars, *29*
  Dark Forces, *31*
  Dead of Alive, *659*
  Diablo, *32*; *446*; *448*
  DOOM, *30*; *31*; *618*; *717*
  Duke Nukem 3D, *31*
  Final Fantasy, *32*
  Flight Simulator, *30*
  FreakOut, *45*
  Galaxian, *655*
  Gazzillonaire, *32*
  Half-Life, *618*
  Hexen, *31*
  Jazz Jackrabbit, *32*
  Loaded, *446*
  Mahjong, *33*
  Marble Madness, *446*
  Monopoly, *33*
  Mortal Kombat, *659*
  Outpost, *801*
  Pac Man, *32*
  PacMan, *649*; *651*; *655*
  PaperBoy, *446*
  Phoenix, *655*

Pong, *29*
Populous, *32*
Quake, *31*; *618*; *677*
Rex Blade, *31*
SimAnt, *32*
SimCity, *32*
Soul Blade, *659*
Tekken, *659*
Tetris, *33*
Unreal, *31*
Warcraft, *32*
Wing Commander, *30*
Wolfenstein 3D, *30*
Zaxxon, *446*; *448*
Zork, *765*
Идентификатор интерфейса, *203*; *226*
Изображение, *100*
Интерфейс, *201*; *219*
Искусственный интеллект, *35*; *647–707*
Генетические алгоритмы, *697*
Запоминание, *668*
Нейронные сети, *695*
Нечеткая логика, *699*
Обучение, *668*
Планирование, *670*
Поиск пути, *677*
Следование за объектом, *650*
Случайное движение, *649*
Уклонение от объекта, *654*

К

Кинематика, *749*
Кисть, *81*; *162*
Клавиатура, *145*; *506–14*
Класс Windows, *77*
Регистрация, *83*
Код символа, *145*
Конечный автомат, *660*
Консольное приложение, *71*
Контекст графического устройства, *158*
Контекст устройства, *78*
Кохена–Сазерленда алгоритм, *381*
Критический раздел, *646*
Курсор, *100*; *105*

Л

Лексический анализ, *776*
Линейный режим, *243*

М

Массив, *594*
Масштабирование, *334*; *402*; *409*
Матрица, *403*
Вектор, *407*
Единичная, *405*
Масштабирования, *409*
Нулевая, *405*
Переноса, *409*
Поворота, *410*
Размерность, *403*
Сложение и вычитание, *405*
Умножение матричное, *406*
Умножение скалярное, *406*
Матричного анализа основы, *403–9*
Меню, *116*
Горячие клавиши, *118*
Каскадные, *117*
Метафайл, *101*
Многозадачность, *65*; *624*
Многопоточность, *65*; *623–46*
Мышь, *151*; *515–19*

Н

Нейронные сети, *695*
Нечеткая логика, *699*

О

Обработчик событий, *86*
Обработчик сообщений, *90*
Окно, *76*
Оптимизация, *611*
Отсечение, *300–313*; *374–87*
Список отсечения, *308*

П

Палитра, *133*; *235*; *322*; *363*
Физическая, *363*
Переключение страниц, *239*; *272*; *280–86*
Перенос, *392*; *409*
Перо, *159*
Пиксель, *130*
Пиктограмма, *100*; *102*
Поверхность
Вторичная, *239*
Поворот, *394*; *410*
Поток, *624*

Прозрачность, *287*
Прокрутка страниц, *439*
Процедура Windows, *86*

Р

Разрешение, *130*
Растеризация, *367*
Рекурсия, *602*
Ресурсы Windows, *99*
  Диалог, *100*
  Звук, *100*; *110*
  Изображение, *100*
  Курсор, *100*; *105*
  Меню, *116*
  Метафайл, *101*
  Пиктограмма, *100*; *102*
  Строки, *100*; *109*

С

Связанный список, *594*
Семантический анализ, *784*
Семафор, *285*; *645*
Синтаксический анализ, *768*; *781*
Синхронизация, *437*
Системы частиц, *753–62*
Скан-код, *145*
Сложная поверхность, *278*
Сообщения Windows, *87*
  WM_ACTIVATE, *140*
  WM_ACTIVATEAPP, *140*
  WM_CHAR, *145*; *146*
  WM_CLOSE, *140*; *141*; *155*
  WM_COMMAND, *121*
  WM_CREATE, *88*
  WM_DESTROY, *88*; *141*; *155*
  WM_KEYDOWN, *145*; *147*
  WM_KEYUP, *145*; *148*
  WM_LBUTTONDBLCLK, *153*
  WM_LBUTTONDOWN, *153*
  WM_LBUTTONUP, *153*
  WM_MBUTTONDBLCLK, *153*
  WM_MOUSEMOVE, *152*
  WM_MOVE, *140*; *144*
  WM_PAINT, *88*; *126*
  WM_QUIT, *96*; *141*
  WM_RBUTTONDBLCLK, *153*
  WM_RBUTTONDOWN, *153*
  WM_RBUTTONUP, *153*
  WM_SIZE, *140*; *142*
  WM_TIMER, *173*
  WM_USER, *155*
  Обработчик, *90*
Список
  Двухсвязный, *599*
  Связанный, *594*
Спрайт, *296*
Столкновение, *430–37*

Т

Таблица поиска цветов, *133*
Таймер, *172*
Трение, *724*
Тригонометрия, *395–97*; *833–36*
Тройная буферизация, *281*

У

Утечка памяти, *859*

Ф

Физическое моделирование, *707–64*
  Время, *709*
  Гравитация, *718*
  Законы сохранения, *717–18*
  Импульс, *716*
  Кинематика, *749*
  Масса, *708*
  Сила, *715*
  Системы частиц, *753–62*
  Скорость, *711*
  Трение, *724*
  Ускорение, *712*
  Центр масс, *711*
Форматы кодирования цвета, *258*
Функция обратного вызова, *79*; *521*

Х

Хронометрирование, *437*

Ц

Цвет
  16-битовый, *258*
  24-битовый, *268*
  32-битовый, *269*
Цветовой ключ, *297*; *327*
Цветовые эффекты, *340*

Анимация, *340*
Мерцание, *340*
Перестановка, *346*
Сдвиг, *346*

Ч

Чересстрочный вывод, *131*

Ш

Шаблонные изображения, *320*
Шаг видеопамяти, *242*
Ширина видеопамяти, *242*

Научно-популярное издание

Андре Ламот

Программирование игр для Windows. Советы профессионала, 2-е издание

Литературный редактор *Т.П. Кайгородова*
Верстка *О.В. Линник*
Художественный редактор *С.А. Чернокозинский*
Корректоры *З.В. Александрова, Л.А. Гордиенко, Л.В. Чернокозинская*

Издательский дом "Вильямс"
101509, г. Москва, ул. Лесная, д. 43, стр. 1
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати

Подписано в печать 24.02.2003. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 69,3. Уч.-изд. л. 47,8.
Тираж 3500 экз. Заказ № .

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, С.-Петербург, Чкаловский пр., 15.